

SISTEMAS DE BANCO DE DADOS

•EDIÇÃO

Ramez Elmasri
Shamkant B. Navathe

Tradução

Marília Guimarães Pinheiro

Cláudio César Canhette, Glenda Cristina Valim Melo,

Claudia Vicei Amadeu e Rinaldo Macedo de Moraes

Revisão Técnica

Luis Ricardo de Figueiredo

Mestre em ciências da computação e doutorando pela USP-Ribeirão Preto

PEARSON

Addison
Wesley

addisonwiley.com.br

EDITORA AFILIADA

São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha Guatemala México Peru Porto Rico Venezuela

Tradução autorizada a partir da edição original em inglês, *Fundamentals of Database Systems* 4th ed., Elmasri, Ramez; Navathe, Shamkant B., publicada pela Pearson Education, sob o selo Addison Wesley.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Pearson Education do Brasil.

Gerente Editorial: Roger Trimer
Editora de Desenvolvimento: Renatha Prado
Gerente de Produção: Heber Lisboa
Editora de Texto: Tereza Gouveia
Preparação: Maria Alice da Costa, Margô Negro e Nelson Luis Barbosa
Revisão: Alessandra Miranda, Alexandra Costa e Thelma Babaoka
Capa: Marcelo Françoze, sobre o projeto original de Beth Anderson
Imagem da Capa © 2003 Digital Vision
Editoração Eletrônica: ERJ Composição Editorial e Artes Gráficas Ltda.

A Figura 12.14 é uma definição de um diagrama do modelo lógico de dados em Rational Rose®.
A Figura 12.15 é um projeto de banco de dados em Rational Rose®. A Figura 12.17 é um diagrama de classe para o banco de dados empresa gerado pela Rational Rose®.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Elmasri, Ramez

Sistemas de banco de dados / Ramez Elmasri e Shamkant B. Navathe; revisor técnico Luis Ricardo de Figueiredo. -- São Paulo : Pearson Addison Wesley, 2005

Título original: Fundamentals of database systems
Vários tradutores.
Bibliografia.

ISBN 85-88639-17-3

1. Banco de dados I. Navathe, Shamkant B. II. Título.

04-6763

CDD-005.75

índices **para** catálogo sistemático

1. Banco de dados : Sistemas : Processamento de dados 005.75
2. Banco de dados : Fundamentos : Processamento de dados 005.75

2006

1ª reimpressão

Direitos exclusivos para a língua portuguesa cedidos à
Pearson Education do Brasil,
uma empresa do grupo Pearson Education
Av. Ermano Marchetti, 1435
CEP: 05038-001 - Lapa - São Paulo - SP
Tel: (11) 2178-8686 Fax: (11) 3611-0444
e-mail: vendas@pearsoned.com

A Amalia com amor R.E.

*A minha mãe, Vijaya, e
minha esposa, Aruna, pelo amor e apoio
S.B.N.*

Prefácio

Este livro introduz os conceitos fundamentais necessários para projetar, usar e implementar os sistemas de banco de dados e suas aplicações. Nossas apresentações abordam com profundidade os fundamentos da modelagem e projeto de bancos de dados, suas linguagens e as funcionalidades dos sistemas de gerenciamento de bancos de dados e as técnicas de implementação desses sistemas. O livro foi projetado para ser usado como um livro-texto para os cursos de um ou dois semestres em sistemas de banco de dados nos níveis introdutório, avançado, de graduação ou de pós-graduação, e como um livro de referência. Pressupomos que os leitores estejam familiarizados com a programação elementar e com os conceitos de estruturação de dados, e que tenham tido algum contato com a organização básica de computadores.

Iniciamos na Parte 1 com uma introdução e uma apresentação dos conceitos básicos e sua terminologia, além dos princípios de modelagem conceitual de banco de dados. E concluímos o livro nas partes 7 e 8 com uma introdução sobre as tecnologias emergentes, como *data mining* (garimpagem de dados), XML, segurança e banco de dados para Web. Nas partes intermediárias — 2 a 6 — tratamos com profundidade os aspectos mais importantes dos fundamentos de banco de dados.

As características fundamentais incluídas nesta edição são:

- O livro possui uma organização auto-ajustável e flexível para que possa ser adaptada a necessidades individuais.
- A abordagem para a modelagem de dados agora inclui o modelo ER e a UML.
- Um novo capítulo de SQL avançado com material sobre as técnicas de programação em SQL, como JDBC e SQL/CLI.
- Dois exemplos utilizados ao longo do livro — chamados EMPRESA e UNIVERSIDADE — permitem ao leitor comparar as diferentes abordagens que usam a mesma aplicação.
- A cobertura de assuntos como segurança, bancos de dados móveis, GIS e gerenciamento de dados Genoma foi atualizada.
- Um novo capítulo em XML e bancos de dados para a Internet.
- Um capítulo novo em *data mining*.
- Uma significativa revisão dos suplementos para incluir um conjunto robusto de materiais para professores e estudantes e um estudo de caso on-line.

Principais Diferenças em Relação à Terceira Edição

Existem várias mudanças organizacionais na quarta edição, além de alguns capítulos novos muito importantes. As mudanças principais são:

- Os capítulos sobre organização de arquivos e indexação (capítulos 5 e 6 na terceira edição) foram movidos para a Parte 4 da atual edição e correspondem agora aos capítulos 13 e 14. Essa parte inclui também os capítulos 15 e 16 sobre processamento de pesquisas (*query*) e sua otimização e o projeto do banco de dados físico e sintonia (*tuning*) (que se referia ao Capítulo 18 e seções 16.3-16.4 da edição anterior).
- A cobertura do modelo relacional foi reorganizada e atualizada na Parte 2 desta edição. O Capítulo 5 cobre os conceitos do modelo relacional e as restrições. O material sobre álgebra relacional e cálculo foi incorporado ao Capítulo 6. O projeto de banco de dados relacional usando os mapeamentos do ER-para-relacional e do EER-para-relacional encontra-se no Capítulo 7. A SQL está nos capítulos 8 e 9, bem como o material novo sobre técnicas de programação em SQL nas seções 9.3 a 9.6.
- A Parte 3 cobre a teoria e a metodologia de projeto de banco de dados. Os capítulos 10 e 11 sobre a teoria da normalização correspondem aos capítulos 14 e 15 da edição anterior. O Capítulo 12, referente ao projeto de banco de dados, foi, na prática, atualizado para incluir mais cobertura sobre a UML.
- Os capítulos sobre transações, controle de concorrência e recuperação (19, 20 e 21 na terceira edição) estão agora nos capítulos 17, 18 e 19 na Parte 5 desta edição.
- Os capítulos que dizem respeito aos conceitos de orientação a objetos, modelo de objetos ODMG e sistemas objeto-relacional (11, 12 e 13 na edição precedente) fazem parte atualmente dos capítulos 20, 21 e 22 na Parte 6. O Capítulo 22 foi reorganizado e atualizado.

- Os capítulos 10 e 17 da terceira edição foram retirados. O material sobre arquitetura cliente-servidor foi incluído nos capítulos 2 e 25 da nova edição.
- Os capítulos sobre segurança, modelos estendidos (ativo, temporal, espacial, multimídia) e bancos de dados distribuídos (capítulos 22, 23 e 24 na edição anterior) correspondem agora aos capítulos 23, 24 e 25 na Parte 7.0 capítulo sobre segurança foi atualizado. O Capítulo 25 da terceira edição relativo a bancos de dados dedutivos foi incorporado ao Capítulo 24 desta edição e faz parte da Seção 24-4.
- O Capítulo 26 é novo, se refere a XML (eXtended Markup Language) e está relacionado ao acesso a bancos de dados relacionais pela Internet.
- O material sobre *data mining* e *data warehousing* (Capítulo 26 na edição anterior) foi dividido em dois capítulos. O Capítulo 27 sobre *data mining* foi ampliado e atualizado.

Conteúdo Desta Edição

A Parte 1 descreve os conceitos básicos necessários para um bom entendimento do projeto de banco de dados e sua implementação, bem como as técnicas de modelagem conceituais usadas nos sistemas de banco de dados. Os capítulos 1 e 2 introduzem os bancos de dados, seus usuários típicos e os conceitos de SGBDs, sua terminologia e arquitetura. No Capítulo 3, os conceitos do modelo Entidade-Relacionamento (ER) e os diagramas ER são apresentados e utilizados para ilustrar o projeto de banco de dados conceitual. O Capítulo 4 está concentrado na abstração de dados e nos conceitos de modelagem de dados semânticos, e estende o modelo ER para incorporar essas idéias, direcionando para o modelo de dados ER-Estendido (EER) e diagramas EER. Os conceitos apresentados incluem subclasses, especialização, generalização e tipos de união (categorias). A notação para os diagramas de classe da UML também foi introduzida nos capítulos 3 e 4-

A Parte 2 descreve o modelo de dados relacional e os SGBDs relacionais. O Capítulo 5 descreve o modelo relacional básico, suas restrições de integridade e operações de atualização. O Capítulo 6 traz as operações da álgebra relacional e introduz o cálculo relacional. O Capítulo 7 discute o projeto de banco de dados relacional utilizando os mapeamentos do ER e do EER-para-relacional. O Capítulo 8 dá uma visão detalhada da linguagem SQL, cobrindo o padrão SQL, que é implementado na maioria dos sistemas relacionais. O Capítulo 9 trata dos tópicos de programação SQL, como SQLJ, JDBC e SQL/CLI.

A Parte 3 cobre vários tópicos relacionados ao projeto de banco de dados. Os capítulos 10 e 11 tratam do formalismo, teorias e algoritmos desenvolvidos para o projeto de banco de dados relacional por meio da normalização. Esse material inclui os tipos de dependências funcionais e outros tipos, e a forma normal de relações. Passo a passo, a normalização intuitiva é apresentada no Capítulo 10, e o projeto de algoritmos relacionais é discutido no Capítulo 11, que também define outros tipos de dependências, como a multivalorada e a junção. O Capítulo 12 apresenta uma visão geral das diferentes fases do projeto de banco de dados para as aplicações de tamanhos médios e grandes, usando UML.

A Parte 4 começa com uma descrição das estruturas de arquivo físicas e dos métodos de acesso usados em sistemas de banco de dados. O Capítulo 13 aborda os métodos primários de organizar os arquivos de registros em disco, incluindo o *hash* estático e o dinâmico. O Capítulo 14 refere-se a técnicas de indexação de arquivos, incluindo as estruturas de dados árvore-B e árvore-B+, bem como arquivos *grid*. O Capítulo 15 introduz os fundamentos do processamento e otimização de pesquisas, e o Capítulo 16 discute o projeto de banco de dados físico e sua sintonização.

A Parte 5 analisa o processamento de transações, controle de concorrência e técnicas de recuperação, incluindo as discussões de como esses conceitos são efetivados na SQL.

A Parte 6 é uma introdução ampla aos sistemas de bancos de dados orientados a objeto e objeto-relacional. O Capítulo 20 insere os conceitos de orientação a objetos. O Capítulo 21 dá uma visão detalhada do modelo de objetos ODMG e suas linguagens associadas ao ODL e OQL. O Capítulo 22 descreve como estão sendo estendidos os bancos de dados relacionais para incluir os conceitos orientados a objeto e apresenta as características do sistema objeto-relacional, além da avaliação de algumas características do padrão SQL3 e o modelo de dados relacional aninhado.

As partes 7 e 8 cobrem vários tópicos avançados. O Capítulo 23 fornece uma visão geral de segurança e autorização em banco de dados, inclusive dos comandos privilegiados da SQL, GRANT (CONCEDER) e REVOKE (REVOGAR), e amplia os conceitos de segurança incluindo a criptografia, os papéis e o controle de fluxo. O Capítulo 24 introduz vários modelos de banco de dados estendidos para aplicações avançadas. Inclui banco de dados ativo e gatilhos (*triggers*), temporal, espacial, multimídia e bancos de dados dedutivos. O Capítulo 25 possui uma introdução a bancos de dados distribuídos e à arquitetura cliente-servidor de três camadas. O Capítulo 26 é novo e trata do modelo XML (eXtended Markup Language). Primeiramente, discute as diferenças entre os modelos estruturado, semi-estruturado e não-estruturado; a seguir, apresenta os conceitos de XML; e, finalmente, compara o modelo XML com os modelos de banco de dados tradicionais. O Capítulo 27 sobre *data mining* foi ampliado e atualizado. O Capítulo 28 introduz os conceitos de *data warehousing*.

Por fim, o Capítulo 29 traz uma

introdução aos tópicos de bancos de dados móveis, bancos de dados multimídia, GIS (Sistemas de Informações Geográficas) e gerenciamento de dados Genoma em bioinformática.

O Apêndice A contém várias alternativas de notações diagramáticas para exibir um esquema conceitual de ER ou EER. Elas podem ser substituídas pelas notações que empregamos, se o professor assim o desejar. O Apêndice C traz os mais importantes parâmetros físicos para os discos. Os apêndices B, E e F estão no site. O Apêndice B é um estudo de caso que acompanha o projeto e a implementação do banco de dados de uma livraria. Os apêndices E e F cobrem os sistemas de bancos de dados legados, baseados nos modelos de bancos de dados de rede e hierárquicos. Foram usados por mais de 30 anos como base para muitas aplicações de bancos de dados comerciais existentes e sistemas de processamento de transações, e serão necessárias várias décadas para substituí-los completamente. Consideramos importante apresentar aos estudantes de administração de banco de dados essas abordagens existentes há muito tempo. Os capítulos completos da segunda edição podem ser encontrados no site como uma referência a esta edição.

Orientações Para o Uso Deste Livro

Existem diferentes formas de apresentação de um curso de banco de dados. Os capítulos das partes 1 a 5 podem ser usados em um curso introdutório de sistemas de banco de dados na ordem em que estão ou na ordem escolhida pelo professor. O professor pode omitir capítulos e seções e incluir outros do restante do livro, dependendo da ênfase do curso. Ao término de cada seção de apresentação do capítulo são apresentadas as seções que podem ser omitidas se for aconselhável uma discussão menos detalhada do tópico abordado no capítulo. Sugerimos o uso até o Capítulo 14 em um curso de banco de dados introdutório e a inclusão de outras partes selecionadas dos demais capítulos, dependendo do conhecimento dos estudantes e da profundidade desejada. Para uma ênfase em técnicas de implementação de sistemas, os capítulos das partes 4 e 5 devem ser incluídos.

Os capítulos 3 e 4, que tratam da modelagem conceitual usando-se os modelos ER e EER, são importantes para uma boa compreensão conceitual sobre os bancos de dados. Porém, podem ser estudados parcialmente, estudados em outro curso ou até mesmo ser omitidos, se a ênfase for a implementação do SGBD.

Os capítulos 13 e 14, referentes à organização de arquivos e indexação, também podem ser estudados posteriormente ou ainda ser omitidos, caso a ênfase seja nos modelos de banco de dados e linguagens. Para os estudantes que já participaram.

.

de um curso em organização de arquivos, partes desses capítulos poderiam ser utilizadas como material de leitura, ou alguns exercícios podem ser definidos como revisão dos conceitos.

Um ciclo de vida completo de projeto e implementação de um banco de dados engloba o projeto conceitual (capítulos 3 e 4), o mapeamento do modelo (Capítulo 7), a normalização (Capítulo 10) e a implementação em SQL (Capítulo 9). A documentação adicional em SGBDRS também deve ser providenciada.

O livro foi escrito de forma a abranger todos esses tópicos, embora sua ordem de leitura possa ser alterada. A figura da página anterior mostra as principais interdependências entre os capítulos. Como a figura ilustra, é possível começar com diferentes tópicos após os dois capítulos introdutórios. Embora a figura possa parecer complexa, é importante observar que, se os capítulos forem lidos na ordem proposta, as interdependências não serão perdidas. A figura pode ser consultada por professores que desejam utilizar uma ordem alternativa de apresentação.

Para o curso de um único semestre baseado neste livro, alguns capítulos podem ser definidos como material de leitura, como as partes 4, 7 e 8. O livro pode ser usado, também, para um caso de dois semestres. O primeiro curso, "Introdução ao projeto de sistemas de bancos de dados", nos níveis introdutório, principiante ou avançado, deve cobrir os capítulos 1 a 14. O segundo, "Técnicas de projeto e implementação de bancos de dados", para níveis mais avançados, deve abordar os capítulos 15 a 28. Os capítulos das partes 7 e 8 podem ser utilizados seletivamente em cada semestre, e outras bibliografias descrevendo os SGBDs, disponíveis para os estudantes em bibliotecas, podem ser empregadas para complementar o material do livro.

Material de apoio (no Companion Website)

No site www.aw.com/elmasri_br professores e estudantes encontram material de apoio exclusivo, incluindo:

- "Estudos de caso" sobre o projeto e a implementação do banco de dados de uma livraria.
- Capítulos 10 e 11 da terceira edição.
- Apêndices E e F (em inglês).
- Conjunto de "Notas de aula" em PowerPoint.
- Manual de soluções (em inglês) exclusivo para professores que lecionam a disciplina.
- Exercício de múltipla escolha.

Agradecimentos

É um grande prazer agradecermos a assistência e contribuição de um grande número de pessoas que nos auxiliaram neste esforço. Primeiro, gostaríamos de agradecer a nossos editores, Maite Suarez-Rivas, Katherine Harutunian, Daniel Rausch e Ju-liet Silveri. Em particular, agradecemos os esforços e a ajuda de Katherine Harutunian, nosso primeiro contato para a quarta edição. Gostaríamos de expressar nosso reconhecimento a todos que tenham contribuído para esta quarta edição. Agradecemos a contribuição dos seguintes revisores: Phil Bernhard, Florida Tech; Zhengxin Chen, University of Nebraska em Omaha; Jan Chomicki, University of Buffalo; Hakan Ferhatosmagnoglu, Ohio State University; Len Fisk, Califórnia State University, Chico; William Hankley, Kansas State University; Ali R. Hurson, Penn State University; Vijay Kumar, University of Missouri-Kansas City; Peretz Shoval, Ben-Gurion University, Israel; Jason T. L. Wan, New Jersey Institute of Technology; e Ed Omiecinski da Geórgia Tech, que contribuiu com o Capítulo 27.

Ramez Elmasri agradece a seus alunos Hyoil Han, Babak Hojabri, Jack Fu, Charley Li, Ande Swathi, e Steven Wu, que contribuíram com material para o Capítulo 26. É grato também ao apoio oferecido pela University of Texas, Arlington.

Sham Navathe agradece a Dan Forsythe e aos seguintes alunos da Geórgia Tech: Weimin Feng, Angshuman Guin, Abrar Ul-Haque, Bin Liu, Ying Liu, Wanxia Xie e Waigen Yee.

Gostaríamos de reforçar nossos agradecimentos àqueles que revisaram e contribuíram para as edições anteriores de *Fundamentos de Sistemas de Banco de Dados*. Na primeira edição, somos gratos a: Alan Apt (editor), Don Batory, Scott Downing, Dennis Heimbinger, Julia Hodges, Yannis Ioannidis, Jim Larson, Dennis McLeod, Per-Ake Larson, Rahul Patel, Nicholas Roussopoulos, David Stemple, Michael Stonebraker, Frank Tompa, e Kyu-Young Whang; na segunda edição: Dan Joraans-tad (editor), Rafi Ahmed, Antônio Albano, David Beech, José Blakeley, Panos Chrysanthis, Suzanne Dietrich, Vic Ghorpa-dey, Goets Graefe, Eric Hanson, Junguk L. Kim, Roger King, Vram Kouramajian, Vijay Kumar, John Lowther, Sanjay Manchanda, Toshimi Minoura, Inderpal Mumick, Ed Omiecinski, Girish Pathak, Raghu Ramakrishnan, Ed Robertson, Eu-gene Sheng, David Stotts, Marianne Winslett, e Stan Zdonick. Agradecemos às pessoas que contribuíram para a terceira edição: nossos editores na Addison-Wesley, Maite Suarez-Rivas, Katherine Harutunian, e Bob Woodbury, e os seguintes colegas que contribuíram ou revisaram parcial ou totalmente a terceira edição: Suzanne Dietrich, Ed Omiecinski, Rafi Ahmed, Fran-cois Bancilhon, José Blakeley, Rick Cattell, Ann Chervenak, David W. Embley, Henry A. Etlinger, Leonidas Fegaras, Dan

Forsyth, Farshad Fötouhi, Michael Franklin, Sreejith Gopinath, Goetz Craefe, Richard HuU, Sushil Jajodia, Ramesh K. Kar-ne, Harish Kotbagi, Vijay Kumar, Tarcisio Lima, Ramon A. Mata-Toledo, Jack McCaw, Dennis McLeod, Rokia Missaoui, Magdi Morsi, M. Narayanaswamy, Carlos Ordonez, Joan Peckham, Betty Salzberg, Ming-Chien Shan, Junping Sun, Rajs-hekhar Sunderraman, Aravindan Veerasamy e Emilia E. Villareal.

Por último, mas nem por isso menos importante, somos muito gratos ao apoio, ao encorajamento e à paciência de nossos familiares.

1

INTRODUÇÃO MODELAGEM CONCEITUAL

À

1

Banco de Dados e os Usuários de Banco de Dados

Os bancos de dados e os sistemas de bancos de dados se tornaram componentes essenciais no cotidiano da sociedade moderna. No decorrer do dia, a maioria de nós se depara com atividades que envolvem alguma interação com os bancos de dados. Por exemplo, se formos ao banco para efetuarmos um depósito ou retirar dinheiro, se fizermos reservas em um hotel ou para a compra de passagens aéreas, se acessarmos o catálogo de uma biblioteca informatizada para consultar uma bibliografia, ou se comprarmos produtos — como livros, brinquedos ou computadores — de um fornecedor por intermédio de sua página Web, muito provavelmente essas atividades envolverão uma pessoa ou um programa de computador que acessará um banco de dados. Até mesmo os produtos adquiridos em supermercados, em muitos casos, atualmente, incluem uma atualização automática do banco de dados que mantém o controle do estoque disponível nesses estabelecimentos.

Essas interações são exemplos do que podemos denominar **aplicações tradicionais de banco de dados**, no qual a maioria das informações que são armazenadas e acessadas apresenta-se em formatos textual ou numérico. Nos últimos anos, os avanços tecnológicos geraram aplicações inovadoras e interessantes dos sistemas de banco de dados.

Os **bancos de dados de multimídia** podem, agora, armazenar figuras, vídeos e mensagens sonoras. Os **sistemas de informações geográficas** (*geographic information systems* — GIS) são capazes de armazenar e analisar mapas, dados do tempo e imagens de satélite. Os ***data warehouses*** e os ***online analytical processing*** (OLAP) — processamento analítico on-line — são utilizados em muitas empresas para extrair e analisar as informações úteis dos bancos de dados para a tomada de decisões.

A **tecnologia de bancos de dados ativos** (*active database technology*) e ***real time*** (tempo real) são usados no controle de processos industriais e de produção (indústria). As técnicas de pesquisa em banco de dados estão sendo aplicadas na World Wide Web para aprimorar a recuperação de informações necessárias pelos usuários da Internet.

Entretanto, para entendermos os fundamentos da tecnologia de banco de dados, devemos começar pelas aplicações tradicionais de bancos de dados. Sendo assim, na Seção 1.1 deste capítulo, definimos o que é um banco de dados e conceituamos alguns termos básicos. Na Seção 1.2 apresentamos um banco de dados como exemplo, uma UNIVERSIDADE, para ilustrar nossa discussão. Em seguida, na Seção 1.3, descrevemos algumas características principais dos sistemas de banco de dados, e nas seções 1.4 e 1.5 categorizamos os tipos de pessoas cujas profissões envolvem o uso e a interação com os sistemas de banco de dados. Nas seções 1.6, 1.7 e 1.8 discutiremos as diversas capacidades de um sistema de banco de dados e algumas aplicações típicas. A Seção 1.9 resume todo o capítulo.

O leitor que optar por uma rápida introdução aos sistemas de banco de dados pode estudar as seções 1.1 a 1.5, depois, pode saltar ou folhear as seções 1.6 até 1.8 e iniciar o Capítulo 2.

Nossa opção foi priorizar, sempre, o termo mais comumente utilizado nas áreas de ensino e pesquisa de banco de dados. Este termo é geralmente empregado em inglês, optamos por essa forma no texto traduzido. Entretanto, muitas vezes traduziremos também o termo para melhor compreensão daqueles que não estão familiarizados com tal terminologia. (N. de T.)

1.1 Introdução

Os bancos de dados e a sua tecnologia estão provocando um grande impacto no crescimento do uso de computadores. É viável afirmar que eles representam um papel crítico em quase todas as áreas em que os computadores são utilizados, incluindo negócios, comércio eletrônico, engenharia, medicina, direito, educação e as ciências da informação, para citar apenas algumas delas. A palavra *banco de dados* é tão comumente utilizada que, primeiro, devemos defini-la. Nossa definição inicial é bastante genérica.

Um **banco de dados** é uma coleção de dados relacionados. Os **dados** são fatos que podem ser gravados e que possuem um significado implícito. Por exemplo, considere nomes, números telefônicos e endereços de pessoas que você conhece. Esses dados podem ter sido escritos em uma agenda de telefones ou armazenados em um computador, por meio de programas como o Microsoft Access ou Excel. Essas informações são uma coleção de dados com um significado implícito, conseqüentemente, um banco de dados.

A definição de banco de dados, mencionada anteriormente, é muito genérica. Por exemplo, podemos considerar o conjunto de palavras que formam esta página como dados relacionados, portanto, constituindo um banco de dados. No entanto, o uso do termo *banco de dados* é geralmente mais restrito. Possui as seguintes propriedades implícitas:

- Um banco de dados representa alguns aspectos do mundo real, sendo chamado, às vezes, de **minimundo** ou de **universo de discurso (UoD)**. As mudanças no minimundo são refletidas em um banco de dados.
- Um banco de dados é uma coleção lógica e coerente de dados com algum significado inerente. Uma organização de dados ao acaso (randômica) não pode ser corretamente interpretada como um banco de dados.
- Um banco de dados é projetado, construído e povoado por dados, atendendo a uma proposta específica. Possui um grupo de usuários definido e algumas aplicações preconcebidas, de acordo com o interesse desse grupo de usuários.

Em outras palavras, um banco de dados possui algumas fontes das quais os dados são derivados, alguns níveis de interação com os eventos do mundo real e um público efetivamente interessado em seus conteúdos.

Um banco de dados pode ser de qualquer tamanho e de complexidade variável. Por exemplo, a lista de nomes e endereços, citada anteriormente, pode possuir apenas poucas centenas de registros, cada um com uma estrutura simples. Porém, o catálogo computadorizado de uma grande biblioteca pode conter meio milhão de entradas organizadas em diferentes categorias — pelo sobrenome principal do autor, pelo assunto, pelo título —, sendo cada categoria organizada em ordem alfabética. Um banco de dados muito maior e mais complexo é mantido pelo Internal Revenue Service (IRS), órgão responsável pelo controle dos formulários de impostos preenchidos pelos contribuintes dos Estados Unidos. Se pressupomos que existam cem milhões de contribuintes e cada um deles preenche em média cinco formulários com aproximadamente 400 caracteres de informações por formulário, teríamos um banco de dados de $100 \times 10^6 \times 400 \times 5$ caracteres (bytes) de informação. Se o IRS mantiver os últimos três formulários para cada contribuinte teremos, além do atual, um banco de dados de 8×10^8 bytes (800 gigabytes). Essa imensa quantidade de informação deve ser organizada e gerenciada para que os usuários possam pesquisar, recuperar e atualizar os dados necessários.

Um banco de dados pode ser gerado e mantido manualmente ou pode ser automatizado (computadorizado). Por exemplo, um catálogo de cartões bibliotecários é um banco de dados que oferece a possibilidade de ser criado e mantido manualmente. Um banco de dados computadorizado pode ser criado e mantido tanto por um grupo de aplicativos escritos especialmente para essa tarefa como por um sistema gerenciador de banco de dados. É claro que, neste livro, o objetivo é abordar os bancos de dados computadorizados.

Um **sistema gerenciador de banco de dados (SGBD)** é uma coleção de programas que permite aos usuários criar e manter um banco de dados. O SGBD é, portanto, um *sistema de software de propósito geral* que facilita os processos de *definição, construção, manipulação e compartilhamento* de bancos de dados entre vários usuários e aplicações. A **definição** de um banco de dados implica especificar os tipos de dados, as estruturas e as restrições para os dados a serem armazenados em um banco de dados.

A **construção** de um banco de dados é o processo de armazenar os dados em alguma mídia apropriada controlada pelo SGBD. A **manipulação** inclui algumas funções, como pesquisas em banco de dados para recuperar um dado específico, atualização do banco para refletir as mudanças no minimundo e gerar os relatórios dos dados. O **compartilhamento** permite aos múltiplos usuários e programas acessar, de forma concorrente, o banco de dados.

Outras funções importantes do SGBD são a *proteção* e a *manutenção* do banco de dados por longos períodos. A **proteção** inclui a *proteção do sistema* contra o mau funcionamento ou falhas (*crashes*) no hardware ou software, e *segurança* contra acessos

1 Usaremos a palavra *dados* (*data*, em inglês) tanto para o singular como para o plural, como é usual na literatura; o contexto determinará a interpretação. No inglês formal, o termo *dados* é utilizado para o plural, e *datum*, para o singular.

não autorizados ou maliciosos. Um banco de dados típico pode ter um ciclo de vida de muitos anos, então, os SGBD devem ser capazes de **manter** um sistema de banco de dados que permita a evolução dos requisitos que se alteram ao longo do tempo.

Não é necessário usar os softwares SGBD típicos para implementar um banco de dados computadorizado. Poderíamos escrever nosso próprio conjunto de programas para criar e manter um banco de dados criando, de fato, nosso próprio SGBD com uma *finalidade específica*. Nesses casos — se usarmos um SGBD de propósito geral ou não —, normalmente teremos de desenvolver uma quantidade considerável de softwares complexos. Na verdade, a maioria dos SGBD é composta por sistemas muito complexos.

Para completar nossa definição inicial chamaremos o banco de dados e o software SGBD, juntos, de **sistema de banco de dados**. A Figura 1.1 ilustra alguns dos conceitos discutidos.

1.2 UM EXEMPLO

Considerando um exemplo simples com o qual a maioria dos leitores está muito familiarizada: um banco de dados de uma UNIVERSIDADE, no qual são mantidas as informações do meio acadêmico, como alunos, cursos e notas. A Figura 1.2 mostra a estrutura do banco de dados e fornece uma pequena amostra dos dados desse banco. O banco é organizado em cinco arquivos, cada um armazena os registros de dados do mesmo tipo. O arquivo ALUNO conserva os dados de cada estudante, o CURSO preserva os dados sobre cada curso, o arquivo DISCIPLINA guarda os dados de cada disciplina do curso. Continuando, o arquivo HISTORICO_ESCOLAR mantém as notas recebidas por aluno nas diversas disciplinas cursadas e, finalmente, o arquivo PRE_REQUISITO armazena os pré-requisitos de cada curso.

Para *definir* esse banco de dados devemos especificar a estrutura de cada registro em cada arquivo, considerando-se os diferentes tipos de elementos **dos** dados a serem armazenados em cada registro. Na Figura 1.2, cada registro ALUNO inclui os dados que representam o NomedoEstudante, NumerodoAluno, Turma (calouro ou 1, veterano ou 2...) e Curso Habilitação (matemática ou mat, ciência da computação ou CC...); cada registro CURSO apresenta dados como NomedoCurso, NumerodoCurso, Créditos e Departamento (que oferece o curso) etc. Precisamos ainda especificar os **tipos de dados** para cada elemento de dados em um registro. Por exemplo, podemos especificar que nome em ALUNO é uma *string* (cadeia) de caracteres alfabéticos, número do aluno em ALUNO é um inteiro (*integer*) e o HISTORICO_ESCOLAR é um caractere único do conjunto {A, B, C, D, F, I}. Podemos

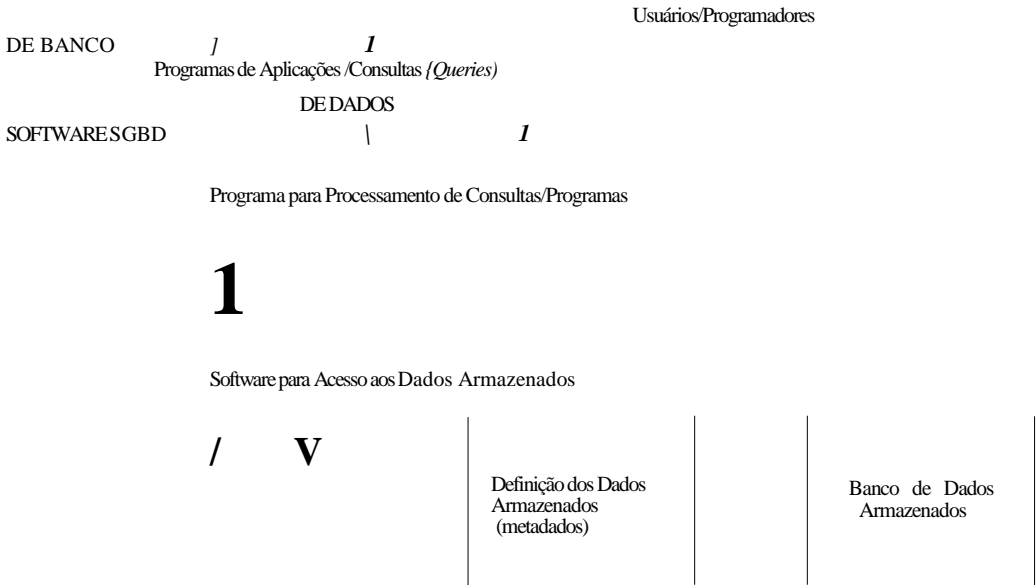


FIGURA 1.1 Configuração de um sistema de banco de dados simplificado.

2 Usamos, aqui, o termo arquivo informalmente. No nível conceitual, *arquivo* é uma *coleção* de registros que podem ou não estar ordenados.

usar ainda um esquema de código para representar os valores de um determinado dado. Por exemplo, na Figura 1.2, representamos a turma do ALUNO por 1 para os calouros; 2 para os veteranos; 3 para os que cursam o penúltimo ano; 4 para aqueles do último ano; e 5 para os alunos graduados.

Para *construir* o banco de dados UNIVERSIDADE armazenamos os dados que representem cada aluno, curso, disciplina, relatório de notas e pré-requisitos, bem como cada registro no arquivo apropriado. Pode-se perceber que os registros de diferentes arquivos podem estar relacionados. Por exemplo, o registro para "Smith" no arquivo ALUNO está relacionado a dois registros no arquivo HISTORICO_ESCOLAR, cuja função é especificar as notas de Smith em duas disciplinas. Por similaridade, cada registro, no arquivo PRE_REQUISITO, está relacionado a dois registros em curso: um representando o curso e o outro, o pré-requisito. A maioria dos bancos de dados médios e grandes inclui muitos tipos de registros e tem *muitos relacionamentos* entre os registros.

A *manipulação* do banco de dados envolve uma consulta (*query*) e atualização. Os exemplos de consulta são: a 'recuperação do histórico escolar — lista de todos os cursos e notas — de Smith', a 'relação dos nomes dos alunos que fizeram as disciplinas do curso de banco de dados oferecido no segundo semestre de 1999 e suas notas naquelas disciplinas' e 'quais os pré-requisitos do curso de banco de dados?' Os exemplos de atualização são: 'mudar a turma de Smith para veteranos', 'criar uma nova disciplina para o curso de banco de dados neste semestre' e 'colocar a nota A para Smith na disciplina banco de dados do último semestre'. Essas consultas informais e atualizações devem ser especificadas, precisamente, na linguagem de consulta (*query language*) do SGBD, antes de serem processadas.

ALUNO	Nome	Numero	Turma	Curso_Hab
	Smith	17	1	CC
	Brown	8	2	CC

NomedoCurso	NumerodoCurso	Créditos	Departamento
Introdução à Ciência da Computação	CC1310	4	CC
Estruturas de dados	CC3320	4	CC
Matemática Discreta	MAT2410	3	MATH
Banco de dados	CC3380	3	CC

DISCIPLINA	IdentificadordeDisciplina	NumerodoCurso	Semestre	Ano	Instrutor
	85	MAT2410	Segundo Semestre	98	Kihg
	92	CC1310	Segundo Semestre	98	Anderson
	102	CC3320	Primeiro Semestre	99	Knuth
	112	MAT2410	Segundo Semestre	99	Chang
	119	CC1310	Segundo Semestre	99	Anderson
	135	CC3380	Segundo Semestre	99	Stone

HISTORICO,,ESCOLAR	NumerodoAluno	Identificador/Disciplinas	Nota
	17	112	B
	17	119	C
	8	85	A
	8	92	A
	8	102	B
	8	135	A

PRE_REQUISITO	NumerodoCurso	NumerodoPre_requisito
	CC3380	CC3320
	CC3380	MAT2410
	CC3320	CC1310

FIGURA 1.2 Um banco de dados que armazena as informações de alunos e cursos.

1.3 CARACTERÍSTICAS DO EMPREGO DE BANCOS DE DADOS

Um número significativo de características distingue a abordagem que utiliza o banco de dados daquela tradicional que usa a programação e arquivos. No tradicional **processamento de arquivos**, cada usuário define e implementa os arquivos necessários para uma aplicação específica, como parte da programação da aplicação. Por exemplo, um usuário, a *secretaria de notas*, pode manter um arquivo para os alunos e suas notas. Os programas para imprimir um histórico do aluno e colocar novas notas no arquivo são implementados como parte da aplicação. Um segundo usuário, o *departamento de contabilidade*, pode controlar os dados relacionados às mensalidades e pagamentos dos alunos. Apesar de ambos os usuários estarem interessados nos dados sobre os estudantes, cada um mantém suas informações em arquivos separados — e os programas que manipulam esses arquivos — porque cada um deles precisa de alguns dados não disponíveis nos arquivos do outro. Essas redundâncias, na definição e armazenamento dos dados, resultam em um espaço de armazenamento desperdiçado e em esforços redundantes para manter os dados comuns atualizados.

Na abordagem utilizando um banco de dados, um único repositório de dados é definido uma única vez, mantido e então acessado por vários usuários. As principais características da abordagem de um banco de dados *versus* a abordagem de processamento de arquivos são as seguintes:

- Natureza autodescritiva do sistema de banco de dados.
- Isolamento entre os programas e os dados, e a abstração dos dados.
- Suporte para as múltiplas visões dos dados.
- Compartilhamento de dados e processamento de transações de multiusuários.

Descreveremos, a seguir, cada característica em seções separadas. As características adicionais dos sistemas de banco de dados serão discutidas nas seções 1.6 a 1.8.

1.3.1 Natureza Autodescritiva do Sistema de Banco de Dados

Uma característica fundamental da abordagem de um banco de dados é que o sistema de banco de dados possui não apenas o banco de dados, mas também uma completa definição ou descrição da estrutura desse banco de dados e suas restrições. Essa definição está armazenada no catálogo do SGBD, que contém informações como a estrutura de cada arquivo, o tipo e o formato de armazenamento de cada item de dado e várias restrições sobre os dados. A informação armazenada no catálogo é chamada **metadados** e descreve a estrutura do banco de dados primário (Figura 1.1).

O catálogo é usado tanto pelo software SGBD como pelos usuários do banco de dados que precisam de informações sobre a estrutura desse banco. Um pacote de software SGBD de propósito geral não está escrito para uma aplicação específica, portanto, será necessário acessar o catálogo para conhecer a estrutura dos arquivos no banco de dados, como o tipo e o formato dos dados que o programa vai acessar. O SGBD precisa trabalhar bem com *qualquer número de aplicações* — por exemplo, um banco de dados de uma universidade, de um banco ou de uma empresa —, desde que a definição do banco de dados esteja armazenada no catálogo.

No processamento tradicional de arquivos, a definição dos dados faz parte dos próprios programas da aplicação. Em consequência disso, esses programas são restritos a trabalhar com um *único banco de dados específico*, cuja estrutura esteja declarada no programa da aplicação. Por exemplo, um software de uma aplicação escrito em C++ pode ter a *struct* ou a declaração de classes, e um programa em COBOL tem comandos na Data Division para definir seus arquivos. Porém, o programa de processamento de arquivos possibilita o acesso a um único banco de dados específico, enquanto o SGBD pode acessar diversos bancos de dados, extraindo as definições de banco de dados do catálogo e usando-as depois.

No exemplo mostrado na Figura 1.2, o catálogo do SGBD armazenará as definições de todos os arquivos mostrados. Elas são especificadas pelo projetista antes de criar o banco de dados real e armazenadas no catálogo. Todas as vezes que um pedido for feito para acessar, digamos, o registro Nome de um ALUNO, o SGBD se referirá ao catálogo para determinar a estrutura do arquivo ALUNO e a posição e tamanho do item de dado Nome dentro do registro ALUNO. Em contraste, em uma aplicação típica de processamento de arquivos, a estrutura do arquivo e a localização exata, no caso extremo, de Nome dentro do registro ALUNO, já estão codificadas em cada programa que acessa esses itens de dados.

1.3.2 Isolamento entre os Programas e Dados e Abstração de Dados

No processamento tradicional de arquivos, a estrutura do arquivo de dados está embutida no programa da aplicação, sendo assim, qualquer mudança na estrutura de um arquivo pode exigir *alterações de todos os programas* que acessam esse arquivo. Ao

contrário, os programas para acesso ao SGBD não exigem essas alterações na maioria dos casos. A estrutura dos arquivos de dados é armazenada no catálogo do SGBD separadamente do programa de acesso. Denominaremos essa propriedade **independência programa-dados**.

Por exemplo, o programa de acesso a arquivos pode ser escrito de forma que acesse, apenas, os registros ALUNO da estrutura apresentada na Figura 1.3. Se quisermos adicionar outro dado ao registro de cada ALUNO, digamos, sua Data de Nascimento, esse programa não vai trabalhar por muito tempo e precisará ser alterado. Ao contrário, em um ambiente SGBD, necessitamos alterar apenas a descrição do registro ALUNO no catálogo para refletir a inclusão do novo item de dados Data de Nascimento; nenhum programa será modificado. A próxima vez que um programa SGBD acessar o catálogo, a nova estrutura do registro ALUNO será acessada e utilizada.

Nome do Item de Dado	Posição Inicial no Registro	Tamanho em Caracteres (bytes)
Nome	1	30
Número do Aluno	31	4
Turma	35	4
Curso_Hab	39	4

FIGURA 1.3 Formato de armazenamento interno para um registro ALUNO.

Em alguns tipos de sistemas de banco de dados, como o orientado a objeto e o objeto-relacional (capítulos 20 a 22), os usuários podem estabelecer as operações sobre os dados como parte das definições de dados. Uma **operação** (também chamada *função* ou *método*) é especificada em duas partes. A *interface* (ou assinatura) de uma operação **inclui** o nome da operação e os tipos de dados de seus argumentos (ou parâmetros). A *implementação* (ou *método*) de uma operação é definida separadamente e pode ser alterada sem afetar a interface. Os programas de usuários da aplicação podem operar nos dados invocando essas operações por meio de seus nomes e argumentos, sem considerar como essas operações são implementadas. Isso pode ser chamado de **independência programa-operação**.

A característica que permite a independência programa-dados e programa-operação é intitulada **abstração de dados**. Um SGBD oferece aos usuários uma **representação conceitual** de dados que não inclui muitos detalhes sobre como o dado é armazenado ou como as operações são implementadas. Informalmente, o **modelo de dados** é um tipo de abstração de dados usado para prover essa representação conceitual. O modelo de dados utiliza os conceitos lógicos, como objetos, suas propriedades e seus inter-relacionamentos, que podem ser mais fáceis para os usuários entenderem os conceitos de armazenamento computacionais. Consequentemente, o modelo de dados *esconde* os detalhes de armazenamento e da implementação, desinteressantes para a maioria dos usuários de banco de dados.

Por exemplo, vamos considerar novamente a Figura 1.2. A implementação interna do arquivo pode ser definida pelo comprimento de seus registros —, o número de caracteres (bytes) em cada registro —, e cada item de dado pode ser especificado pelo seu byte inicial dentro de um registro e seu comprimento em bytes. O registro ALUNO poderia, em razão disso, ser representado como exposto na Figura 1.3. No entanto, um usuário típico de banco de dados não está preocupado com a localização de cada item de dados dentro de um registro ou com o seu comprimento; na realidade, sua preocupação é que quando for acessado o Nome do ALUNO, o valor correto seja retornado. Uma representação conceitual dos registros ALUNO é mostrada na Figura 1.2. Muitos outros detalhes da organização do armazenamento de dados, como os caminhos de acesso especificados em um arquivo, podem ser escondidos dos usuários de banco de dados pelo SGBD — discutiremos os detalhes do armazenamento nos capítulos 13 e 14.

Na abordagem de banco de dados, a estrutura detalhada e a organização de cada arquivo são armazenadas no catálogo. Os usuários de banco de dados e os programas de aplicação referem-se à representação conceitual dos arquivos, e o SGBD extrai os detalhes do armazenamento de arquivos do catálogo, quando são necessários, pelos módulos de acesso a arquivos do SGBD. Muitos modelos de dados podem ser utilizados para prover essa abstração dos dados aos usuários do banco. A maior parte deste livro é dedicada à apresentação dos vários modelos de dados e os conceitos que estes utilizam para abstrair a representação dos dados.

Nos bancos de dados orientados a objeto e a objeto-relacional, o processo de abstração não inclui apenas a estrutura de dados, mas também as operações sobre os dados. Essas operações oferecem uma abstração das atividades do minimundo facilmente entendidas pelos usuários.

Por exemplo, uma operação de CALCULO_GPA pode ser aplicada ao objeto ALUNO para calcular a média de pontos nas notas. Essas operações podem ser invocadas pela consulta do usuário ou pelos programas de aplicação sem ter de se saber os detalhes de como as operações são implementadas. Nesse sentido, uma abstração de uma atividade de um minimundo está disponível para o usuário como uma **operação abstrata**.

Um banco de dados típico tem muitos usuários, e cada qual pode solicitar diferentes perspectivas ou visões do banco de dados. Uma visão pode ser um subconjunto de um banco de dados ou conter uma visão virtual dos dados, derivados dos arquivos do banco de dados, mas não, explicitamente, armazenados. Alguns usuários podem não saber se os dados a que eles se referem são armazenados ou derivados.

Um SGBD multiusuários, cujos usuários têm várias aplicações distintas, deve proporcionar facilidades para a definição de múltiplas visões. Por exemplo, um usuário do banco de dados da Figura 1.2 pode estar interessado somente em acessar e imprimir o histórico de cada aluno; a visão para esse usuário é mostrada na Figura 1.4a. Um segundo usuário, interessado em checar se os alunos cumpriram todos os pré-requisitos de cada curso para o qual se matricularam, pode utilizar a visão apresentada na Figura 1.4b.

HISTORICO_ESCOLAR	NomedoAluno	Histórico Escolar do Aluno				
		NumerodoCurso	Nota	Semestre	Ano	IdDisciplina
	Smith	CC1310	C	Outono	99	119
		MAT2410	B	Outono	99	112
	Brown	MAT2410	A	Outono	98	85
		CC1310	A	Outono	98	92
		CC3320	B	Primavera	99	102
		CC3380	A	Outono	99	135

PRE.REQUISITOS	NomedoCurso	NumerodoCurso	Pre_Requisitos
	Banco de Dados	CC3380	CC3320
			MAT2410
	Estruturas de Dados	CC3320	CC1310

FIGURA 1.4 Duas visões derivadas de um banco de dados da Figura 1.2. (a) Visão do HISTÓRICO ESCOLAR DO ALUNO, (b) Visão dos PRÉ-REQUISITOS DO CURSO.

1.3.4 Compartilhamento de Dados e o Processamento de Transação Multiusuários

Um SGBD multiusuário, como o nome implica, deve permitir que diversos usuários acessem o banco de dados ao mesmo tempo. Isso é essencial se os dados para as várias aplicações estão integrados e mantidos em um único banco de dados. O SGBD deve incluir um software de controle de concorrência para garantir que muitos usuários, ao tentar atualizar o mesmo dado, o façam de um modo controlado, para assegurar que os resultados das atualizações sejam corretos. Por exemplo, quando muitos atendentes tentam reservar um lugar em um voo, o SGBD deve garantir que cada assento possa ser acessado somente por um atendente de cada vez, para fazer a reserva de apenas um passageiro. Esses tipos de aplicações são, normalmente, denominados aplicações de processamento de transações on-line — *online transaction processing* (OLTP). Uma regra fundamental do software do SGBD multiusuário é garantir que as transações concorrentes operem corretamente.

O conceito de transação tornou-se fundamental para muitas aplicações de banco de dados. Uma transação é um *programa em execução* ou *processo* que inclui um ou mais acessos ao banco de dados, como a leitura ou a atualização de registros. Cada transação deve executar um acesso logicamente correto ao banco de dados, se executado sem a interferência de outras transações. O SGBD deve garantir várias propriedades da transação. A propriedade de isolamento garante que cada transação possa ser efetuada de forma isolada de outras transações; mesmo centenas de transações podem ser executadas simultaneamente. A propriedade de atomicidade garante que todas as operações em um banco de dados, em uma transação, sejam executadas ou nenhuma delas o seja. Discutiremos as transações em detalhes na Parte V do livro.

As características precedentes são muito importantes para distinguir um SGBD de um software tradicional de processamento de arquivos. Na Seção 1.6 abordaremos as funcionalidades adicionais que caracterizam um SGBD. Primeiro, no entanto, categorizaremos os diferentes tipos de pessoas que trabalham em um ambiente de sistema de banco de dados.

1.4 ATORES NO PALCO

Para um pequeno banco de dados pessoal, como a agenda de endereços discutida na Seção 1.1, uma pessoa em geral define, constrói e manipula um banco de dados — não há compartilhamento. No entanto, muitas pessoas estão envolvidas no projeto, uso e manutenção de um grande banco de dados com centenas de usuários. Nesta seção, identificaremos as pessoas cujas profissões envolvem o dia-a-dia do uso de um grande banco de dados; nós as chamaremos de 'atores no palco'. Na Seção 1.5 consideraremos as pessoas que podem ser nomeadas 'trabalhadores dos bastidores', ou seja, aqueles que trabalham para manter o ambiente dos sistemas de banco de dados, mas que não estão interessados no banco de dados de fato.

141 Administradores de Banco de Dados

Em uma organização na qual muitas pessoas usam os mesmos recursos, há a necessidade de um administrador-chefe para gerenciar esses recursos. No ambiente de banco de dados, o principal recurso é o próprio banco de dados e, a seguir, o SGBD e os softwares relacionados. Administrar esses recursos é responsabilidade do **administrador de banco de dados** — *database administrator* (DBA). O DBA é o responsável pela autorização para o acesso ao banco, pela coordenação e monitoração de seu uso e por adquirir recursos de software e hardware conforme necessário. O DBA é o responsável por problemas como brechas de segurança ou tempo de resposta ruim do sistema. Em grandes organizações, o DBA possui suporte de assistentes que o auxiliam no desempenho dessas funções.

142 Os Projetistas do Banco de Dados

Os projetistas do banco de dados são responsáveis pela identificação dos dados que serão armazenados no banco e também por escolher as estruturas apropriadas para representar e armazenar esses dados. Essas tarefas são, em sua maioria, realizadas antes que o banco de dados seja realmente implementado e alimentado com os dados. Ainda é de responsabilidade desse profissional

comunicar-se antecipadamente com todos os prováveis usuários do banco para conhecer suas necessidades (requisitos) e criar projetos que as atendam. Em alguns casos, os projetistas estão na equipe do DBA e podem executar outras tarefas, depois que o projeto do banco de dados estiver completo.

Os projetistas de banco de dados normalmente interagem com os potenciais grupos de usuários e desenvolvem visões (views) do banco de dados que atendam aos requisitos de dados e ao processamento desses grupos. Cada visão é, então, analisada e *integrada* às visões de outros grupos de usuários. O projeto final do banco de dados deve ser capaz de suportar todos os requisitos de todos os grupos de usuários.

143 O Usuário Final

Os **usuários finais** são pessoas cujas profissões requerem o acesso a um banco de dados para consultas, atualização e relatórios; o banco de dados existe prioritariamente para os seus usuários. Há várias categorias de usuários finais:

- **Usuários finais casuais:** acionam o banco de dados ocasionalmente, mas precisam de informações diferentes a cada acesso. Eles usam uma linguagem de consulta a banco de dados sofisticada para especificar suas solicitações e normalmente são gerentes de nível médio ou elevado ou outros profissionais com necessidades ocasionais.
- **Iniciantes ou usuários finais parametrizáveis:** compõem uma grande parcela dos usuários finais de banco de dados. Seu trabalho exige constante envolvimento com consulta e atualização de um banco de dados, usando tipos de consulta e atualizações padronizadas — chamadas **transações enlatadas** — que tenham sido cuidadosamente programadas e testadas. As tarefas que esses usuários desempenham são variadas:
 - Os caixas de banco checam os saldos das contas e informam as retiradas e os depósitos.
 - Os funcionários responsáveis pela reserva de vôos, hotéis e locação de carros checam a viabilidade para atender às solicitações de reservas e as confirmam.
 - Os funcionários em agências de correio informam as identificações de pacotes por códigos de barra e informações descritivas para atualizar um banco de dados central de pacotes recebidos e em trânsito.
- **Usuários finais sofisticados:** incluem os engenheiros, cientistas, analistas de negócios e outros que se familiarizam com as facilidades do SGBD para implementar aplicações que atendam às suas solicitações complexas.

Em português, o termo mais comum são transações customizadas. (N. de T.).

1.5 Trabalhadores dos Bastidores

- **Usuários autônomos (*stand-alone*):** mantêm um banco de dados pessoal por meio do uso de pacotes de programas prontos que possuem interfaces gráficas ou programas baseados em menus fáceis de usar. Um exemplo disso é o usuário de um pacote para cálculo de impostos que armazena seus dados financeiros pessoais para o pagamento dos impostos.

Um SGBD típico provê múltiplas facilidades de acesso ao banco de dados. Os usuários iniciantes precisam aprender muito pouco sobre as facilidades oferecidas pelo SGBD; eles têm de entender somente as interfaces das transações-padrão projetadas e implementadas para seu uso. Os usuários casuais aprendem apenas as poucas facilidades que utilizam repetidamente. Os usuários sofisticados tentam aprender a maioria das funcionalidades do SGBD com o objetivo de executar suas necessidades complexas. Os autônomos se tornam muito proficientes no uso de pacotes de softwares específicos.

1.4.4 Analistas de Sistemas e Programadores de Aplicações (Engenheiros de Software)

Os **analistas de sistemas** determinam as solicitações dos usuários finais, especialmente os usuários finais iniciantes e os parametrizáveis, além de desenvolver as especificações das transações customizadas que atendam a essas solicitações. Os **programadores de aplicações** implementam essas especificações como programas, então eles testam, documentam e mantêm essas transações customizadas. Esses analistas e programadores — também conhecidos como **engenheiros de software** — precisam estar familiarizados com toda a gama de capacidades oferecidas pelo SGBD para realizar suas tarefas.

1.5 TRABALHADORES DOS BASTIDORES

Em adição àquelas pessoas que projetam, usam e administram um banco de dados, outras estão associadas ao projeto, desenvolvimento e operação do *programa e ambiente do sistema* do SGBD. Essas pessoas não têm interesse no banco de dados; são os chamados 'trabalhadores dos bastidores' e incluem as seguintes categorias:

- **Projetistas e implementadores de sistemas SGBD:** são pessoas que projetam e implementam os módulos e interfaces do SGBD, como um pacote. Um SGBD é um sistema com programas complexos com muitos componentes ou **módulos**, incluindo aqueles para implementar o catálogo, processamento de linguagem de consulta (*query language*), interfaces, acesso e armazenamento temporário (*buffering*) dos dados, controle de concorrência, manuseio de recuperação de dados e segurança. O SGBD deve interagir com outros sistemas, tais como o sistema operacional e compiladores de diferentes linguagens de programação.
- **Desenvolvedores de ferramentas:** são pessoas que projetam e implementam as **ferramentas** — os pacotes de programas que facilitam o projeto e uso de um sistema de banco de dados e que ajudam a aprimorar seu desempenho. Essas ferramentas estão em um pacote opcional, adquirido separadamente. Incluem pacotes para projetos de banco de dados, monitoramento de desempenho, interface gráfica ou linguagem natural, protótipo, simulação e geração de dados para teste. Na maioria dos casos, os vendedores de software independentes desenvolvem e negociam essas ferramentas.
- **Pessoal de manutenção e operadores:** são pessoas da administração do sistema, responsáveis pela execução e manutenção do ambiente de hardware e software do sistema de banco de dados.

Embora esses trabalhadores sejam fundamentais para tornar os sistemas de banco de dados disponíveis para os usuários finais, eles normalmente não usam o banco de dados para fins pessoais.

1.6 VANTAGENS DA UTILIZAÇÃO DA ABORDAGEM SGBD

Nesta seção, discutiremos as vantagens da utilização e as funcionalidades que um bom SGBD deve possuir. Elas vão além das quatro características principais abordadas na Seção 1.3. O DBA deve usar essas capacidades para atingir os objetivos relacionados ao projeto, administração e uso de um grande banco de dados multiusuário.

1.6.1 Controle de Redundância

No desenvolvimento tradicional de software utilizando processamento de arquivos, cada usuário mantém seus próprios arquivos para suas aplicações de processamento de dados. Consideremos, por exemplo, o banco de dados UNIVERSIDADE, mencionado na Seção 1.2; aqui, os dois grupos de usuários são os da secretaria do curso e os da contabilidade. Na abordagem tradicional, cada grupo mantém seus arquivos de alunos de maneira independente. A contabilidade também guarda os dados de matrícula e as informações relacionadas a pagamentos, enquanto a secretaria mantém o controle dos cursos e notas dos

Capítulo 1 Banco de Dados e os Usuários de Banco de Dados 12

alunos. Muitos dos dados são armazenados duas vezes: uma vez no arquivo de cada grupo. Outros grupos de usuários podem duplicar alguns ou todos os dados novamente em seus próprios arquivos.

Essa **redundância** em armazenar os mesmos dados várias vezes gera muitos problemas. Primeiro, há a necessidade de desempenhar uma única atualização lógica — como a entrada de dados de novos alunos — várias vezes: uma para cada arquivo no qual o dado do aluno estará armazenado. Isso gera uma *duplicação de esforços*. Segundo, o *espaço de armazenamento é desperdiçado* quando o mesmo dado é armazenado repetidamente; esse problema pode ser sério para os bancos de dados grandes. Terceiro, há a possibilidade de os arquivos que representam os mesmos dados se tornarem *inconsistentes*. Isso pode ocorrer porque uma atualização é aplicada somente a alguns arquivos, mas não em outros. Até mesmo se uma atualização — como a adição de novos alunos — fosse aplicada a todos os arquivos apropriados, os dados dos alunos poderiam, ainda, se tornar *inconsistentes*, porque as atualizações são realizadas independentemente por grupo de usuários. Por exemplo, um grupo de usuários insere a data de nascimento de um aluno incorretamente, como 19-JAN-1984, enquanto outro grupo de usuários pode entrar com a informação correta: 29-JAN-1984.

Na abordagem de banco de dados, as visões de diferentes grupos de usuários são integradas durante o projeto do banco. Idealmente, devemos ter um projeto do banco para armazenar cada item lógico do dado — como o nome do aluno ou a data de nascimento — em *um único lugar* no banco de dados. Isso vai garantir a consistência e economizar espaço de armazenamento. Entretanto, na prática, algumas vezes é necessário o uso de **redundância controlada** para melhorar a performance das consultas. Por exemplo, podemos armazenar NomeDoAluno e o NumeroDoCurso, redundantemente, em um arquivo de HISTORICOESCOLAR (Figura 1.5a), pois quando consultamos um registro HISTORICO_ESCOLAR, queremos o nome do aluno e o número do curso, como também a nota, o número do aluno e o identificador de disciplina. Colocando todos os dados juntos, não temos de pesquisar múltiplos arquivos para coletá-los. Nesse caso, o SGBD deve ter a capacidade de *controlar* essas redundâncias, impedindo as inconsistências entre os arquivos. Isso pode ser feito automaticamente verificando se os valores de NomeDoAluno-NumeroDoAluno para qualquer registro HISTORICO_ESCOLAR na Figura 1.5a possuem correspondentes para Nome-Numero DoAluno nos registros ALUNO (Figura 1.2). De maneira similar, os valores IdentificadorDisciplinas-NumeroDoCurso em historico_escolar podem ser checados com os registros DISCIPLINA.

HISTORICOESCOLAR	NumerodoAluno	NomedoAluno	IdentificadordeDisciplina	NumerodoCurso	Nota
	17	Smith	112	MAT2410	B
	17	Smith	119	CC1310	C
	8	Brown	85	MAT2410	A
	8	Brown	92	CC1310	A
	8	Brown	102	CC3320	B
	8	Brown	135	CC3320	A

HISTORICO_ESCOLAR	NumerodoAluno	NomedoAluno	IdentificadordeDisciplina	NumerodoCurso	Nota
	17	Brown	112	MAT2410	B

FIGURA 1.5 Armazenamento redundante do NomeDoAluno e NumeroDoCurso no HISTÓRICO ISCOLAR. (a) Dados consistentes, (b) Registro inconsistente.

Alguns desses testes podem ser especificados para o SGBD durante o projeto do banco de dados e automaticamente forçados pelo SGBD, sempre que o arquivo RELATORIO_DE_NOTAS for atualizado. A Figura 1.5b mostra o registro de RELATORIO_DE_NOTAS

inconsistente com o arquivo ALUNO da Figura 1.2, que pode ter sido inserido erroneamente se a redundância *não for controlada*.

1.6.2 Restringindo Acesso Não Autorizado

Quando vários usuários utilizam um grande banco de dados, é provável que a maioria desses usuários não seja autorizada a acessar todas as informações disponíveis no banco de dados. Por exemplo, os dados financeiros são geralmente considerados confidenciais e, por essa razão, somente pessoas com permissão poderão ter acesso a eles. Além disso, a alguns usuários é permitido, apenas, consultar; outros podem consultar e atualizar os dados. Em consequência disso, o tipo de operação de acesso — consulta ou atualização — também deve ser controlado. O mais comum é fornecer aos usuários ou grupo de usuários contas protegidas por senhas, utilizadas para acessar o banco de dados. O SGBD deve garantir a **segurança e um subsistema de autorização** usado pelo DBA para criar contas e definir as restrições de cada uma. O SGBD deve, então, garantir essas restrições automaticamente. Como se pode ver, podemos aplicar os controles semelhantes para o próprio SGBD. Por exemplo, apenas ao

assistente de DBA é permitido o uso de certos **softwares privilegiados**, como o utilizado para criar novas contas. Da mesma forma, os usuários parametrizáveis podem ter permissão para acessar o banco de dados somente por meio de transações específicas, desenvolvidas para seu uso.

1.63 Garantindo o Armazenamento Persistente para Objetos Programas

Os bancos de dados podem ser usados para oferecer um **armazenamento persistente** aos objetos programas e estruturas de dados. Essa é uma das principais justificativas para os **sistemas de banco de dados orientados a objeto**. As linguagens de programação têm uma estrutura de dados complexa, como os tipos de registro em Pascal ou as definições de classe em C++ ou Java. Os valores das variáveis dos programas são descartados, uma vez que o programa termina sua execução, a não ser que o programador os armazene, explicitamente, em arquivos permanentes, os quais, normalmente, envolvem a conversão de estruturas complexas em um formato adequado para o armazenamento em arquivos. Quando surge a necessidade de ler os dados mais uma vez, o programador deve convertê-los do formato de arquivo para uma estrutura variável do programa. Os sistemas de banco de dados orientado a objeto são compatíveis com as linguagens de programação como C++ e Java, e o software SGBD, automaticamente, executa qualquer conversão necessária. Conseqüentemente, um objeto complexo em C++ pode ser armazenado permanentemente em um SGBD orientado a objeto. Esse objeto é conhecido como **persistente**, desde que exista após o término de execução dos programas e possa, depois, ser acessado por outro programa em C++.

O armazenamento persistente de programas e as estruturas de dados são uma importante função do sistema de banco de dados. Os sistemas tradicionais de banco de dados geralmente possuem o chamado **problema de separação por impedância**, quando as estruturas de dados fornecidas pelo SGBD são incompatíveis com as estruturas de dados da linguagem de programação. Os sistemas de banco de dados orientados a objeto oferecem estruturas de dados **compatíveis** com uma ou mais linguagens de programação orientadas a objeto.

1.64 Garantindo o Armazenamento de Estruturas para o Processamento Eficiente de Consultas

Os sistemas de banco de dados devem fornecer funcionalidades para *a execução de atualizações e consultas eficientemente*. Pelo fato de o banco de dados ser armazenado, tipicamente, em disco, o SGBD deve possuir estruturas de dados especializadas para aumentar a velocidade de pesquisa no disco dos registros desejados. Os arquivos auxiliares, chamados **indexes (indexados)**, são utilizados com esse objetivo. Os **indexes** são baseados em estruturas de dados árvores (tree) ou estruturas de dados *hash*, adequadamente adaptados para a pesquisa em disco. Com o intuito de processar os registros necessários do banco de dados para uma consulta particular, aqueles registros devem ser copiados do disco para a memória. Conseqüentemente, os SGBD em geral têm um módulo de **armazenamento temporário (buffering)** que mantém partes do banco de dados armazenado na memória principal. Em outros casos, o SGBD pode utilizar o sistema operacional para fazer o armazenamento temporário dos dados no disco.

O módulo do SGBD para o **processamento de consulta e otimização** é responsável pela escolha eficiente do plano de execução da consulta (*query*) baseado nas estruturas de armazenamento existentes. A opção de qual *index* criar e manter é parte do *projeto físico do banco de dados* e seu *ajuste (tuning)*, que é de responsabilidade do DBA e sua equipe.

1.65 Garantindo Backup e Restauração

Um SGBD deve prover facilidades para a restauração de falhas de hardware ou de software. O subsistema de **backup e recuperação dos subsistemas** do SGBD é responsável pela recuperação dessas falhas. Por exemplo, se um sistema de computador falhar no meio de uma transação complexa de atualização, o subsistema de recuperação é responsável por garantir que o banco de dados seja recolocado no mesmo estado em que estava, antes do início da execução da transação. Alternativamente, o subsistema pode assegurar que a transação seja resumida do ponto em que foi interrompida — sendo assim, seu efeito completo seria armazenado no banco de dados.

1.66 Fornecendo Múltiplas Interfaces para os Usuários

Como diversos tipos de usuários com níveis de conhecimento técnico diferentes utilizam o banco de dados, o SGBD deve fornecer interfaces diferentes para esses usuários. Essas interfaces incluem linguagens de consulta para os usuários casuais; interfaces de linguagens de programação para programadores de aplicações; formulários e seqüências de comandos para usuários

parametrizáveis; interfaces de menus, interfaces de linguagem natural para usuários autônomos. Ambas, as interfaces com menus e aquelas com formulários, são comumente conhecidas como **interfaces gráficas para os usuários — Graphical User Interfaces (GUIs)**. Muitos ambientes e linguagens especializadas existem para a especificação de GUIs. As capacidades para gerar interfaces Web GUI para um banco de dados — ou capacitando um banco de dados para a Web (*Web-enabling*) — também são muito comuns.

1.6.7 Representando Relacionamentos Complexos entre os Dados

Um banco de dados pode incluir uma grande variedade de dados que estão inter-relacionados de muitas maneiras. Veja o exemplo mostrado na Figura 1.2. O registro de Brown no arquivo ALUNO está relacionado com quatro registros no arquivo HISTORICO_ESCOLAR. De forma similar, cada registro Disciplina está relacionado com um registro de cursos e ainda com os registros HISTORICO_ESCOLAR, um para cada aluno que completou uma disciplina. O SGBD deve ter a capacidade de representar a variedade de relacionamentos complexos entre os dados, bem como recuperar e atualizar os dados relacionados fácil e eficientemente.

1.6.8 Forçando as Restrições de Integridade

A maioria das aplicações de um banco de dados tem certas **restrições de integridade** que devem complementar os dados. O SGBD deve prover funcionalidades para a definição e a garantia dessas restrições. O tipo mais simples de restrição de integridade envolve a especificação de um tipo de dado para cada item de dados. Por exemplo, na Figura 1.2, podemos especificar que o valor do item de dados Turma, em cada registro ALUNO, tem de ser um inteiro entre 1 e 5, e o valor de Nome deve ser uma *string* (cadeia de caracteres) menor que 50 caracteres alfabéticos. Os tipos mais complexos de restrições podem ocorrer, com frequência, envolvendo a definição de que o registro em um arquivo deve estar relacionado aos registros de outros arquivos. Por exemplo, na Figura 1.2, podemos especificar que 'todo registro de disciplina deve estar relacionado com um registro de Curso'. Outro tipo de restrição especifica a singularidade no valor do item de dado, como 'todo registro de curso deve ter um único valor para NumeroDoCurso'. Essas restrições são derivadas do significado ou da **semântica** dos dados e do minimundo que representam. Os projetistas do banco de dados são responsáveis por identificar as restrições de integridade durante o projeto do banco. Algumas restrições podem ser especificadas para o SGBD e executadas automaticamente. Outras podem ser testadas pelos programas de atualização ou no momento da entrada dos dados.

Um item de dados pode ser inserido incorretamente e ainda assim satisfazer as restrições de integridade definidas. Por exemplo, se um aluno recebe nota A, mas é inserida, no banco de dados, a nota C, o SGBD não pode descobrir esse erro, automaticamente, porque C é um valor válido para os tipos de dados de NOTA. Esse erro pode ser descoberto manualmente (quando o aluno receber a nota e reclamar), sendo corrigido, depois, por meio da atualização do banco de dados. Porém, a nota Z pode ser rejeitada automaticamente pelo SGBD, pois ela é um valor inválido para os tipos de dados de NOTA.

1.6.9 Permitindo Inferências e Ações Usando as Regras

Alguns sistemas de banco de dados oferecem capacidades para definir as *regras de dedução* por *inferência* gerando novas informações de fatos armazenados no banco de dados. Esses sistemas são chamados **sistemas de banco de dados dedutivos**. Por exemplo, podem existir regras complexas no minimundo da aplicação para determinar quando um aluno está em recuperação. Isso pode ser especificado *declaradamente* como uma **regra**, e quando for compilada e mantida pelo SGBD, poderá determinar todos os alunos em recuperação. No SGBD tradicional, um *código explícito de programa procedural* teria de ser escrito para dar suporte a essas aplicações. No entanto, se as regras do minimundo mudarem, geralmente é mais conveniente mudar as regras de dedução declaradas que reescrever os programas procedurais. Os **sistemas de banco de dados ativos** oferecem funcionalidades mais potentes, pois permitem regras ativas que podem disparar automaticamente ações quando certos eventos e condições ocorrerem.

1.6.10 Implicações Adicionais do Uso da Abordagem de um Banco de Dados

Esta seção discute algumas implicações adicionais da utilização da abordagem do uso de um banco de dados que beneficiam a maioria das organizações:

- **Potencial para Garantir Padrões.** A abordagem de um banco de dados permite ao DBA definir e forçar o uso de padrões entre os usuários de um banco de dados em uma grande organização. Isso facilita a comunicação e cooperação entre os vários departamentos, projetos e os usuários na organização. Os padrões podem ser definidos para os nomes e

formato dos elementos de dados, formatos de exibição, estruturas de relatórios, terminologia, dentre outros. O DBA pode forçar o uso dos padrões em um ambiente de banco de dados centralizado mais facilmente do que em um ambiente no qual cada grupo de usuário tenha o controle de seus próprios arquivos e softwares.

- **Redução no Tempo de Desenvolvimento de Aplicações.** O principal argumento de venda para o uso da abordagem de um banco de dados é que o desenvolvimento de uma nova aplicação — como a recuperação de dados de um banco de dados para a impressão de um novo relatório — demanda um tempo muito pequeno. Projetar e implementar um novo banco de dados do zero deve levar mais tempo do que escrever uma simples aplicação de arquivo especializada. No entanto, uma vez que o banco de dados está ativo e em execução, certamente menos tempo será necessário para criar aplicações usando as facilidades do SGBD. O tempo estimado de desenvolvimento utilizando o SGBD está entre 1/6 e 1/4 do tempo gasto com o sistema tradicional de arquivos.
- **Flexibilidade.** Pode ser necessário alterar a estrutura do banco de dados quando os requisitos mudam. Por exemplo, um novo grupo de usuários pode surgir e precisar de informações não disponíveis no banco de dados. A solução pode ser adicionar um novo arquivo ao banco ou estender os elementos de dados em um arquivo existente. Um SGBD moderno permite certos tipos de alterações evolutivas que mudam a estrutura do banco de dados sem afetar os dados armazenados e os programas de aplicação existentes.
- **Disponibilidade para Atualizar as Informações.** Um SGBD disponibiliza o banco de dados para todos os usuários. Imediatamente após ser feita a atualização de um usuário, todos os outros usuários poderão vê-la. Essa disponibilidade de atualização da informação é essencial para muitas aplicações de processamento de transações, como sistemas de reserva ou banco de dados bancários. É possível fazê-la por intermédio do subsistema de controle de concorrência e recuperação do SGBD.
- **Economias de Escala.** A abordagem do SGBD permite a consolidação dos dados e das aplicações, reduzindo, dessa forma, a perda por superposição entre as atividades de processamento de dados pessoais em diferentes projetos ou departamentos. Isso possibilita que a organização faça investimentos em processadores mais potentes, dispositivos de armazenamento ou equipamentos de comunicação do que cada departamento adquirir seu próprio (menos potente) equipamento, o que reduz o custo total da operação e gerenciamento.

1.7 UMA BREVE HISTÓRIA DAS APLICAÇÕES DE UM BANCO DE DADOS

Apresentaremos, agora, uma breve visão histórica das aplicações que utilizam o SGBD e como elas forneceram a motivação para o surgimento de novos tipos de sistemas de banco de dados.

1.7.1 Primeiras Aplicações de Bancos de Dados Usando Sistemas Hierárquicos e de Rede

A maioria das aplicações pioneiras utilizando um banco de dados mantinha os registros das grandes organizações, como as corporações, universidades, hospitais e bancos. Em muitas dessas aplicações existia um grande número de registros de estruturas semelhantes. Por exemplo, em uma aplicação para uma universidade, as informações similares seriam mantidas para cada aluno, cada curso, cada registro de nota, e assim por diante. Havia também muitos tipos de registros e diversos **inter-relacionamentos** entre eles.

Um dos principais problemas com os sistemas de banco de dados pioneiros era a mistura entre os relacionamentos conceituais, o armazenamento físico e a localização de registros no disco. Por exemplo, o registro de notas de um aluno específico poderia ser fisicamente armazenado próximo ao registro de aluno. Apesar de prover acessos muito eficientes, destinados a consultas originais e transações, para as quais o banco de dados foi projetado para executar, ele não oferecia flexibilidade suficiente e eficiente para os acessos a registros quando novas consultas e transações fossem necessárias. Em especial, as novas consultas que precisavam de uma organização diferente de armazenamento para a eficiência no processamento eram muito difíceis de serem implementadas. Também era muito complicado reorganizar o banco de dados quando as mudanças eram feitas para atender a novos requisitos da aplicação.

Outra deficiência desses sistemas era que forneciam somente as interfaces para a linguagem de programação. Isso fez com que o tempo consumido fosse significativamente alto para implementar novas consultas e transações, pois os novos programas tinham de ser escritos, testados e depurados. A maioria desses sistemas de banco de dados foi implementada em computadores grandes (*mainframes*) e caros, começando em meados de 1960 indo até os anos 70 e 80. Os principais tipos desses sistemas iniciais foram baseados em três paradigmas principais: os sistemas hierárquicos, aqueles baseados em modelo de rede, e os de arquivos invertidos.

1.7.2 Obtendo Flexibilidade de Aplicação com o Banco de Dados Relacional

Os bancos de dados relacionais foram originalmente projetados com o objetivo de separar o armazenamento físico dos dados da sua representação conceitual e prover uma fundamentação matemática para os bancos de dados. O modelo de dados relacional também introduziu as linguagens de consulta de alto nível, que são uma alternativa às interfaces para as linguagens de programação; conseqüentemente, ficou mais rápido escrever novas consultas. A representação de dados relacionais de algum modo assemelha-se ao exemplo apresentado na Figura 1.2. Os sistemas relacionais foram, a princípio, direcionados para as mesmas aplicações dos sistemas pioneiros, mas foi decisivo o fato de oferecerem flexibilidade para o desenvolvimento rápido de novas consultas e para reorganizar o banco de dados quando os requisitos eram alterados.

Os primeiros sistemas relacionais experimentais desenvolveram-se no fim dos anos 70 e os SGBDRs (sistemas de gerenciamento de banco de dados relacional) introduzidos no início dos anos 80 eram muito lentos, pois não usavam ponteiros para o armazenamento físico ou registros de localização para acessar os registros de dados relacionados. Com o desenvolvimento de novas técnicas de armazenamento e indexação, e com o processamento aprimorado de consultas e otimização, seu desempenho melhorou. Assim, os bancos de dados relacionais tornaram-se os tipos dominantes de sistemas para as aplicações tradicionais de banco de dados. Os bancos de dados relacionais agora existem na maioria dos computadores, desde aqueles de uso pessoal até os de grandes servidores.

1.7.3 Aplicações Orientadas a Objeto e a Necessidade de Bancos de Dados Mais Complexos

O aparecimento das linguagens de programação orientadas a objeto nos anos 80 e a necessidade de armazenar e partilhar os objetos complexos estruturados conduziram ao desenvolvimento dos bancos de dados orientados a objeto. Inicialmente, foram considerados como competidores dos bancos de dados relacionais, pois possuíam estruturas de dados mais gerais. Também incorporaram muitos paradigmas úteis orientados a objeto, como tipos de dados abstratos, encapsulamento de operações, herança e identidade de objeto. No entanto, a complexidade do modelo e a falta de um padrão inicial contribuíram para seu uso limitado. Hoje são usados principalmente em aplicações especializadas, tais como projetos em engenharia, publicidade multimídia e sistemas para a indústria.

1.7.4 Trocando Dados na Web para o Comércio Eletrônico (*E-commerce*)

A World Wide Web gerou uma grande rede de computadores interconectados. Os usuários podem criar documentos usando uma linguagem de publicação na Web, como a HTML (Hypertext Markup Language), e armazenar esses documentos nos servidores da Web, na qual os outros usuários (clientes) podem ter acesso. Os documentos podem ser conectados através de *hyperlinks*, que são ponteiros para os outros documentos. Nos anos 90, o comércio eletrônico (*e-commerce*) surgiu como a principal aplicação da Web. Tornou-se rapidamente visível que partes das informações do comércio eletrônico nas páginas Web eram, muitas vezes, dados extraídos dinamicamente dos SGBDs. Várias técnicas foram desenvolvidas para permitir o intercâmbio de dados na Web. Atualmente, a XML (extended Markup Language) é considerada o principal padrão para o intercâmbio de dados entre os vários tipos de banco de dados e as páginas Web. A XML combina os conceitos de modelos empregados nos sistemas de documentos com os conceitos de modelos de bancos de dados.

1.7.5 Ampliando as Funcionalidades dos Bancos de Dados para as Novas Aplicações

O sucesso dos sistemas de banco de dados em aplicações tradicionais encorajou os desenvolvedores de outros tipos de aplicações a se esforçarem para usá-los. Essas aplicações tradicionalmente usavam seus próprios arquivos especializados e estruturas de dados. A seguir, alguns exemplos dessas aplicações:

- Aplicações **científicas**, que armazenam uma grande quantidade de dados resultantes de experimentos científicos em áreas como física avançada ou mapeamento do genoma humano.
- Armazenamento e restauração de imagens, de notícias escaneadas ou fotografias pessoais a imagens fotografadas por satélite e imagens de procedimentos médicos, como raios X ou ressonância magnética.
- Armazenamento e recuperação de vídeos, como filmes ou videoclipes de notícias ou de máquinas fotográficas digitais.
- Aplicações para **data mining** (garimpagem de dados), que analisam grandes quantidades de dados pesquisando as ocorrências de padrões específicos ou relacionamentos.

1.8 Quando Não Usar o SGBD 17

- Aplicações espaciais, que armazenam as informações espaciais dos dados, tais como informações a respeito do tempo ou sobre os mapas usados em sistemas de informações geográficas.
- Aplicações referentes a séries temporais, que guardam informações como os dados econômicos em intervalos regulares de tempo, como, por exemplo, vendas diárias ou composição mensal do produto interno bruto.

Percebeu-se rapidamente que os sistemas relacionais básicos não eram adequados para muitas dessas aplicações, por uma ou mais das razões a seguir:

- As estruturas de dados mais complexas eram necessárias para a modelagem da aplicação do que uma representação relacional simples.
- Os novos tipos de dados eram imprescindíveis, além dos tipos numéricos básicos e as cadeias de caracteres (*strings*).
- As novas operações e construtores de linguagens de consulta eram essenciais para manipular os novos tipos de dados.
- As novas estruturas de armazenamento e indexação eram necessárias.

Isso direcionou os profissionais envolvidos com o desenvolvimento dos SGBDs a adicionar funcionalidades aos seus sistemas. Algumas delas eram de propósito geral, como o conceito de incorporação dos bancos de dados orientados a objeto aos sistemas relacionais. Outras funcionalidades eram específicas, na forma de módulos opcionais que poderiam ser usados para aplicações específicas. Por exemplo, os usuários poderiam comprar uma série de módulos temporais para usar com seus SGBDs relacionais nas suas aplicações com as séries temporais.

1.8 QUANDO NÃO USAR O SGBD

Apesar das vantagens no uso do SGBD, há algumas situações em que esse sistema pode envolver custos altos e desnecessários, que normalmente não ocorreriam no processamento tradicional de arquivos. Os altos custos de utilizar o SGBD são devidos a:

- Investimentos iniciais altos em hardware, software e treinamento.
- Generalidade que o SGBD fornece para a definição e processamento dos dados.
- Custos elevados para oferecer segurança, controle de concorrência, recuperação e funções de integridade.

Problemas adicionais podem surgir se os projetistas do banco de dados e o DBA não projetarem o banco de dados de maneira adequada ou se a aplicação não for implementada apropriadamente. Sendo assim, pode-se indicar o uso de arquivos convencionais nas seguintes circunstâncias:

- O banco de dados e suas aplicações são simples, bem definidas e sem previsão de mudanças.
- Há requisitos de tempo real (*real-time*) para alguns programas difíceis de serem atendidos por causa da sobrecarga (*overhead*) do SGBD.
- O acesso de múltiplos usuários aos dados não é necessário.

1.9 RESUMO

Neste capítulo, definimos um banco de dados como uma coleção de dados relacionados, na qual os *dados* significam fatos registrados. Um típico banco de dados representa alguns aspectos da vida do mundo real e é utilizado por um ou vários grupos de usuários para propostas específicas. Um SGBD é um pacote de software para a implementação e manutenção de bancos de dados computadorizados. O banco de dados e o software, juntos, formam um sistema de banco de dados. Identificamos várias características que distinguem a abordagem de um banco de dados de uma aplicação tradicional de processamento de arquivos. Em seguida, discutimos as categorias principais de usuários do banco de dados ou os 'atores no palco'. Notamos, ainda, que, além dos usuários, há muitas categorias de pessoal de suporte ou os 'trabalhadores dos bastidores', em um ambiente de banco de dados.

Depois, apresentamos a lista de capacidades que devem ser oferecidas por um software SGBD para o DBA, projetistas de bancos de dados e os usuários para ajudá-los a projetar, administrar e usar um banco de dados. A seguir, fizemos um breve histórico da evolução das aplicações de bancos de dados. Finalmente, discutimos os altos custos do uso do SGBD e algumas situações nas quais pode não ser vantajoso utilizar o SGBD.

Questões de Revisão

- 1.1 Defina os seguintes termos: *dados*, *banco de dados*, *SGBD*, *sistema de banco de dados*, *catálogo de banco de dados*, *independência programada*, *visão do usuário* (-user view), *DBA*, *usuário final*, *transação enlatada* (customizada), *sistema de banco de dados dedutivo*, *objeto persistente*, *metadados* e *aplicação de processamento de transação*.
- 1.2 Quais são os três tipos principais de ações que envolvem um banco de dados? Discuta brevemente cada um deles.
- 1.3 Discuta as características principais da abordagem de um banco de dados e como ela difere dos sistemas tradicionais de arquivos.
- 1.4 Quais são as responsabilidades do DBA e dos projetistas de banco de dados?
- 1.5 Quais são os diferentes tipos de usuários finais de banco de dados? Discuta as atividades principais de cada um.
- 1.6 Discuta as funcionalidades que podem ser fornecidas por um SGBD.

Exercícios

- 1.7 Identifique algumas consultas informais e operações de atualizações que você poderia aplicar ao banco de dados exposto na Figura 1.2.
- 1.8 Qual a diferença entre a redundância controlada e aquela sem controle? Ilustre com exemplos.
- 1.9 Nomeie todos os relacionamentos entre os registros do banco de dados apresentados na Figura 1.2.
- 1.10 Apresente algumas visões adicionais para o banco de dados visto na Figura 1.2 que podem ser necessárias para outros grupos de usuários.
- 1.11 Cite alguns exemplos de restrições de integridade que você poderia ter no banco de dados mostrado na Figura 1.2.

Bibliografia Seleccionada

Em outubro de 1991, a edição de *Communications of the ACM* e Kim (1995) apresentou vários artigos descrevendo a próxima geração dos SGBDs; diversas características de um banco de dados, discutidas na época, estão totalmente disponíveis. A edição de março de 1976 da *ACM Computing Surveys* apresenta uma introdução pioneira aos sistemas de banco de dados e fornece uma perspectiva histórica para o leitor interessado.

2

Sistemas de Bancos de Dados: Conceitos e Arquitetura

A arquitetura dos pacotes SGBDs (sistemas gerenciadores de bancos de dados) evoluiu dos sistemas pioneiros e monolíticos, em que o pacote de softwares SGBD era um bloco único formando um sistema fortemente integrado, para os modernos pacotes SGBDs modulares por projeto, com uma arquitetura cliente/servidor. Essa evolução reflete as tendências na computação, na qual os computadores grandes e centralizados (mainframes) estão sendo substituídos por vários PCs (computadores pessoais) e *workstations* (estações de trabalho) conectados via redes de comunicações a vários tipos de servidores — servidores Web, servidores de bancos de dados, servidores de arquivos, servidores de aplicações e assim por diante.

Em uma estrutura básica de SGBD cliente/servidor, as funcionalidades do sistema são distribuídas entre dois tipos de módulos. O **módulo cliente** é projetado para ser executado em uma estação de trabalho ou em um computador pessoal. Em geral, os programas de aplicação e as interfaces de usuário, que acessam o banco de dados, são processados no módulo cliente. Conseqüentemente, o módulo cliente trata a interação com os usuários e oferece uma interface amigável, como formulários ou GUIs (interfaces gráficas para os usuários) baseadas em menus. Outro tipo de módulo, chamado **módulo servidor**, trata de armazenamento de dados, acessos, pesquisas e outras funções. Discutiremos as arquiteturas cliente/servidor detalhadamente na Seção 2.5. Primeiro, estudaremos os conceitos mais básicos que nos propiciarão melhor entendimento das arquiteturas modernas de banco de dados.

Neste capítulo apresentaremos a terminologia e os conceitos básicos utilizados neste livro. Começaremos, na Seção 2.1, discutindo os modelos de dados e definindo os conceitos de esquemas e instâncias, fundamentais para o estudo de sistemas de banco de dados. Discutiremos, então, a arquitetura de SGBD de três esquemas e a independência dos dados na Seção 2.2, o que oferece uma perspectiva para o usuário do que o SGBD deve fazer. Na Seção 2.3 descreveremos os tipos de interfaces e linguagens oferecidas normalmente por um SGBD. Na Seção 2.4 veremos o ambiente de programas dos sistemas de bancos de dados. A Seção 2.5 apresenta uma visão geral dos vários tipos de arquiteturas cliente/servidor. Finalmente, na Seção 2.6 mostraremos uma classificação dos tipos de pacotes SGBD. A Seção 2.7 resume o capítulo.

O material, nas seções 2.4 a 2.6, exibe os conceitos mais detalhados que deverão ser vistos como um suplemento ao material básico introdutório.

2.1 MODELOS DE DADOS, ESQUEMAS E INSTÂNCIAS

Uma característica fundamental do uso de bancos de dados é que permitem a abstração dos dados, ocultando detalhes do armazenamento de dados, que são desnecessários para a maioria dos usuários de bancos de dados. Um **modelo de dados** — conjunto de conceitos que podem ser usados para descrever a estrutura de um banco de dados — fornece o significado necessário

1 Como veremos na Seção 2.5, existem variações da arquitetura cliente/servidor de *duas camadas*.

para permitir essa abstração. Por *estrutura de um banco de dados* entendemos os tipos de dados, relacionamentos e restrições que devem suportar os dados. A maioria dos modelos também inclui uma série de **operações básicas** para a recuperação e atualizações no banco de dados.

Além das operações básicas fornecidas pelo modelo de dados, está se tornando comum incluir conceitos para especificar o **aspecto dinâmico** ou o **comportamento** de uma aplicação de banco de dados. Isso permite que o projetista do banco de dados especifique uma série de **operações definidas pelos usuários**, que são permitidas nos objetos do banco de dados. Um exemplo de uma operação definida pelo usuário poderia ser CALCULEJMPA, que pode ser aplicado a um objeto ALUNO. Porém, as operações genéricas para inserir, excluir, modificar ou pesquisar qualquer tipo de objeto são, freqüentemente, incluídas nas *operações básicas do modelo de dados*. Os conceitos para especificar o comportamento são fundamentais para os modelos de dados orientados a objeto (capítulos 20 e 21), mas também estão sendo incorporados aos modelos de dados tradicionais. Por exemplo, os modelos objeto-relacional (Capítulo 22), que estendem o modelo relacional tradicional, visando incluir esses conceitos, dentre outros.

2.1.1 Categorias de Modelos de Dados

Vários modelos de dados têm sido propostos. Iremos classificá-los de acordo com os tipos de conceitos usados para descrever a estrutura do banco de dados. Os de **alto nível**, ou **modelos de dados conceituais**, possuem conceitos que descrevem os dados como os usuários os percebem, enquanto os de **baixo nível**, ou **modelos de dados físicos**, contêm conceitos que descrevem os detalhes de como os dados estão armazenados no computador. Os conceitos provenientes dos modelos de dados de baixo nível geralmente são significativos para os especialistas em computadores, mas não o são para os usuários finais. Entre esses dois extremos está uma classe de **modelos de dados representacionais** (ou de **implementação**), que oferece os conceitos que podem ser entendidos pelos usuários finais, mas não excessivamente distantes da forma como os dados estão organizados dentro do computador. Os modelos de dados representacionais ocultam alguns detalhes de armazenamento de dados, porém, podem ser implementados em um sistema de computador de maneira direta.

Os modelos de dados conceituais utilizam conceitos como entidades, atributos e relacionamentos. Uma **entidade** representa um objeto do mundo real ou um conceito, como um funcionário ou um projeto, que são descritos no banco de dados. Um **atributo** corresponde a alguma propriedade de interesse que ajuda a descrever uma entidade, como o nome do funcionário ou seu salário. O **relacionamento** entre duas ou mais entidades mostra uma associação entre estas: por exemplo, um relacionamento trabalha-em de um funcionário com um projeto. O Capítulo 3 apresenta o modelo entidade relacionamento — um modelo de dados conceitual de alto nível muito utilizado. O Capítulo 4 descreve os conceitos adicionais para modelagem conceitual de dados, como generalização, especialização e categorias.

Os modelos de dados representacionais ou de implementação são os mais usados nos SGBDs comerciais tradicionais. Incluem o popular **modelo de dados relacional**, bem como os chamados modelos de dados legados — os **modelos de rede** e os **modelos hierárquicos** — amplamente usados no passado. A Parte 2 deste livro é dedicada ao modelo de dados relacional, suas operações e linguagens, bem como a algumas das técnicas para a programação de aplicações em bancos de dados relacionais. O padrão SQL para os bancos de dados relacionais é descrito nos capítulos 8 e 9. Os modelos representacionais de dados representam os dados por meio do uso de estruturas de registro, conseqüentemente, são também chamados **modelos de dados baseados em registro**.

Podemos considerar os **modelos de dados orientados a objeto** uma nova família de modelos de dados de implementação de mais alto nível, muito próximos aos modelos de dados conceituais. Descreveremos as características gerais dos bancos de dados orientados a objeto e o padrão ODMG nos capítulos 20 e 21. Os modelos de dados orientados a objeto são freqüentemente utilizados, ainda, como modelos conceituais de alto nível, especialmente na área de engenharia de software.

Os modelos de dados físicos descrevem como os dados estão armazenados em arquivos no computador, pela representação da informação como o formato do registro, a ordem dos registros e as rotas de acesso (*access paths*). Uma **rota de acesso** é uma estrutura que torna a pesquisa por determinados registros no banco de dados eficiente. Discutiremos as técnicas de armazenamento físico e as estruturas de acesso nos capítulos 13 e 14.

- 2 Algumas vezes, a palavra *modelo* é empregada para denotar uma descrição específica de um banco de dados ou esquema, por exemplo, 'o modelo de dados de marketing'. Não usaremos essa interpretação.
- 3 A inclusão de conceitos que descrevem o comportamento reflete uma tendência, segundo a qual as atividades de projeto de software e do banco de dados estão, progressivamente, sendo combinadas em uma única atividade. Tradicionalmente, a definição do comportamento está associada ao projeto de software.

2.1.2 Esquemas, Instâncias e Estado do Banco de Dados

Em qualquer modelo de dados é importante distinguir entre a *descrição* do banco de dados e o *banco de dados de fato*. A descrição do banco de dados é intitulada **esquema do banco de dados**. Este é definido durante o projeto do banco de dados e não se espera que seja alterado frequentemente. A maioria dos modelos de dados apresenta algumas convenções para a exibição de esquemas como diagramas. Um esquema apresentado é chamado de **diagrama esquemático**. A Figura 2.1 mostra um diagrama esquemático para o banco de dados da Figura 1.2; o diagrama apresenta a estrutura de cada tipo de registro, mas não as instâncias reais dos registros. Chamamos cada objeto no esquema — como um ALUNO ou CURSO — de um **construtor do esquema**.

Um diagrama esquemático mostra somente *alguns aspectos* do esquema, como os nomes dos tipos de registro e itens de dados, além de alguns tipos de restrições. Os outros aspectos não são especificados no diagrama esquemático. Por exemplo, a Figura 2.1 não exibe o tipo de dado de cada item de dado nem os relacionamentos entre os diversos arquivos. Vários tipos de restrições não são representados nos diagramas. Uma restrição como 'alunos do curso de ciência da computação devem cursar a disciplina CC1310 antes do fim do segundo ano' é muito difícil de representar.

O dado no banco de dados pode ser alterado frequentemente. Por exemplo, o banco de dados da Figura 1.2 muda todas as vezes que adicionamos um aluno ou registramos uma nova nota para um aluno. Os dados no banco de dados, em um determinado momento, são chamados de **estado do banco de dados** ou **instantâneo** (*snapshot*). Também chamado o conjunto corrente de **ocorrências** ou **instâncias** no banco de dados. Em um dado estado do banco de dados, cada construtor do esquema tem seu próprio *conjunto corrente* de instâncias; por exemplo, o construtor ALUNO conterá o conjunto de entidades individuais de alunos (registros) como suas instâncias. Muitos estados de bancos de dados podem ser construídos para corresponder a um esquema de banco de dados particular. Cada vez que inserirmos ou deletarmos um registro ou alterarmos o valor de um item de dados em um registro, mudaremos de um estado do banco de dados para outro.

A distinção entre o esquema de banco de dados e o estado de banco de dados é muito importante. Quando **definimos** um novo banco de dados, especificamos o seu esquema de banco de dados somente para o SGBD.

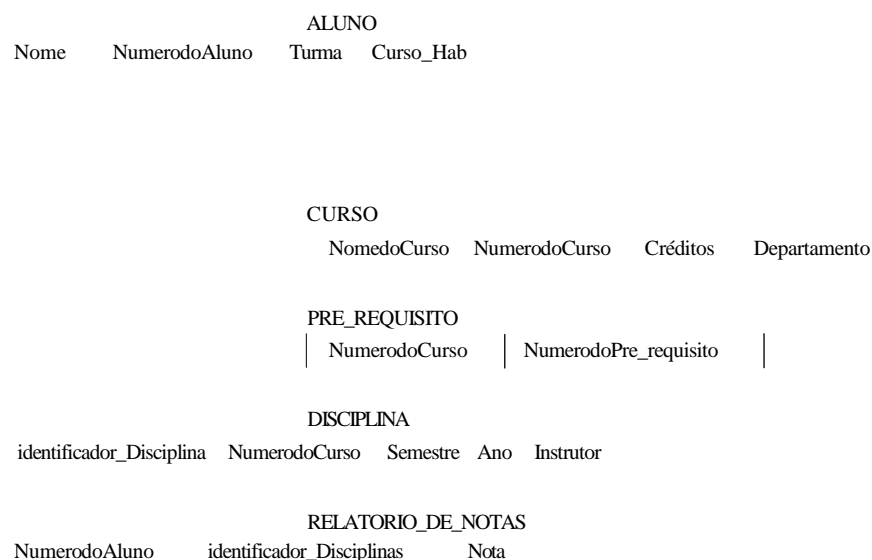


FIGURA 2.1 Diagrama esquemático para o banco de dados da Figura 1.2.

Nesse ponto, o estado correspondente de banco de dados é o *estado vazio*, sem nenhum dado. Teremos o estado inicial do banco de dados quando este for **populado** ou **carregado** com os dados iniciais. A partir daí, todas as vezes que uma operação de atualização for aplicada ao banco de dados, teremos outro estado de banco de dados. A qualquer ponto no tempo, o banco de dados terá um *estado corrente*. O SGBD é parcialmente responsável por assegurar que *cada* estado do banco de dados seja um **estado válido** — isto é, um estado que satisfaz a estrutura e as restrições definidas no esquema. Conseqüentemente, especificar um esquema correto para o SGBD é extremamente importante, e o seu projeto deve ser elaborado com o máximo

As mudanças nos esquemas são necessárias quando os requisitos das aplicações de bancos de dados se modificam. Entre os novos sistemas de bancos de dados estão as operações permitindo as mudanças no esquema, embora o processo de mudança de esquema seja mais complexo que uma simples atualização dos bancos de dados.

É comum na terminologia da área de banco de dados utilizar *esquemas* (seriemos) como o plural de *esquema* (*schema*), ainda que formalmente devesse ser *schemata* (em inglês). A palavra *projeto* (*scheme*) é empregada algumas vezes para representar um esquema. O estado corrente é também chamado de *snapshot* corrente.

cuidado. O SGBD armazena as descrições dos construtores do esquema e suas restrições — também chamadas **metadados** — no catálogo do SGBD, então, o software do SGBD pode referir-se ao esquema quando necessário. O esquema é também chamado **conotação** (*intension*), e o estado do banco de dados, uma **extensão** do esquema.

Embora, conforme mencionado anteriormente, não seja previsto que o esquema seja alterado com frequência, não é raro que mudanças necessitem ser aplicadas ao esquema, quando os requisitos da aplicação forem alterados. Por exemplo, podemos decidir que outro item de dados precise ser armazenado para cada registro em um arquivo, como colocar a *Data de Nascimento* no esquema ALUNO na Figura 2.1. Isso é conhecido como **evolução do esquema**. A maioria dos SGBDs modernos inclui algumas operações para a evolução do esquema, que podem ser aplicadas enquanto o banco de dados está em operação.

2.2 ARQUITETURA DE TRÊS-ESQUEMAS E A INDEPENDÊNCIA DE DADOS

Três das quatro características importantes da abordagem com uso de banco de dados, listados na Seção 1.3, são: 1) separação de programas e dados (independência de dados e operação de programas); 2) suporte a múltiplas visões (*views*) de usuários, e 3) uso de catálogo para armazenar a descrição do banco de dados (esquema). Nesta seção, definiremos uma arquitetura para os sistemas de banco de dados, chamada **arquitetura de três-esquemas**, proposta para auxiliar a realização e visualização dessas características. Depois disso, aprofundaremos a discussão sobre a independência dos dados.

2.2.1 A Arquitetura de Três-Esquemas

O objetivo da arquitetura de três-esquemas, ilustrada na Figura 2.2, é separar o usuário da aplicação do banco de dados físico. Nessa arquitetura, os esquemas podem ser definidos por três níveis:

1. O **nível interno** tem um **esquema interno**, que descreve a estrutura de armazenamento físico do banco de dados. Esse esquema utiliza um modelo de dado físico e descreve os detalhes complexos do armazenamento de dados e caminhos de acesso ao banco de dados.

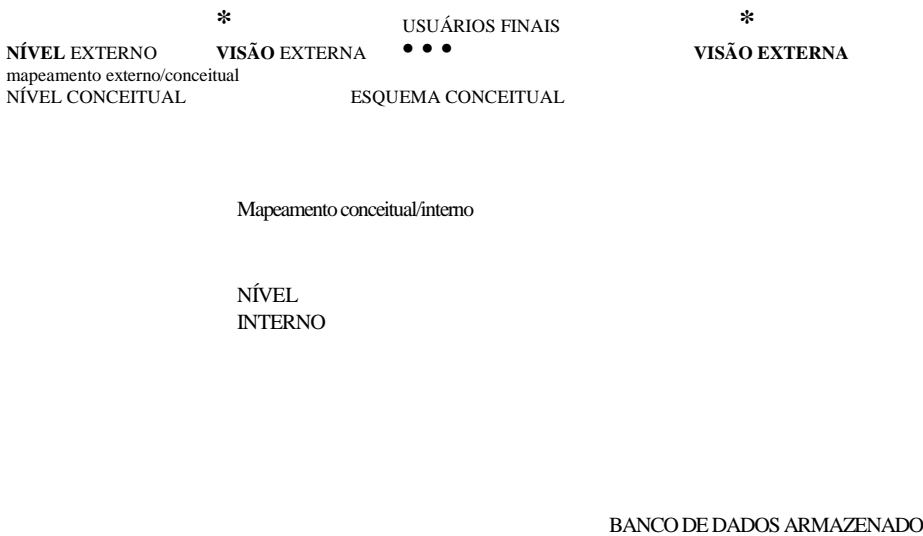


FIGURA 2.2 A arquitetura de três-esquemas.

7 Também conhecida como arquitetura ANSI

/SPARC, segundo o comitê que a propôs (Tsichritzis e Klug 1978).

2. O **nível conceitual** possui um **esquema conceitual**, que descreve a estrutura de todo o banco de dados para a comunidade de usuários. O esquema conceitual oculta os detalhes das estruturas de armazenamento físico e se concentra na descrição de entidades, tipos de dados, conexões, operações de usuários e restrições. Geralmente, um modelo de dados representacional é usado para descrever o esquema conceitual quando o sistema de banco de dados for implementado. Esse *esquema de implementação conceitual* é normalmente baseado em um *projeto de esquema conceitual* em um modelo de dados de alto nível.
3. O **nível externo ou visão** (*view*) abrange os **esquemas externos** ou **visões de usuários**. Cada esquema externo descreve a parte do banco de dados que um dado grupo de usuários tem interesse e oculta o restante do banco de dados desse grupo. Como no item anterior, cada esquema externo é tipicamente implementado usando-se um modelo de dados representacional, possivelmente baseado em um projeto de esquema externo em um modelo de dados de alto nível.

A arquitetura de três-esquemas é uma ferramenta conveniente com a qual o usuário pode visualizar os níveis do esquema em um sistema de banco de dados. A maioria dos SGBDs não separa os três níveis completamente, mas suporta a arquitetura de três-esquemas de alguma forma. Alguns SGBDs incluem detalhes do nível físico no esquema conceitual. Na maioria dos SGBDs que oferece suporte a diferentes visões de usuários, os esquemas externos são especificados no mesmo modelo de dados que descreve a informação no nível conceitual. Alguns SGBDs permitem que diferentes modelos de dados sejam usados nos níveis conceitual e externo.

Observe que os três esquemas são apenas *descrições* dos dados; na verdade, o dado que existe *de fato* está no nível físico. Em um SGBD baseado na arquitetura de três-esquemas, cada grupo de usuários refere-se somente ao seu próprio esquema externo. Consequentemente, o SGBD deve transformar uma solicitação definida no esquema externo em uma solicitação do esquema conceitual, para, então, transformá-la em uma solicitação do esquema interno, a fim de processar o banco de dados armazenado. Se a solicitação for uma recuperação, o dado extraído do banco de dados armazenado deve ser reformatado para adaptar-se à visão externa do usuário. Os processos de transformação de solicitações e resultados entre os níveis são chamados **mapeamentos**. Podem ser consumidores de tempo, sendo assim, alguns SGBDs — especialmente aqueles responsáveis pelo suporte a pequenos bancos de dados — não suportam visões externas. Contudo, até mesmo nesses sistemas, uma certa quantidade de mapeamento é necessária para transformar solicitações entre os níveis conceitual e interno.

2.2.2 Independência de Dados

A arquitetura de três-esquemas pode ser usada para explicar melhor o conceito de **independência de dados**, que pode ser definido como a capacidade de mudar o esquema em um nível do sistema de banco de dados sem que ocorram alterações do esquema no próximo nível mais alto. É possível definir, ainda, dois tipos de independência de dados:

1. **Independência de dados lógica:** é a capacidade de alterar o esquema conceitual sem mudar o esquema externo ou os programas. Podemos modificar o esquema conceitual para expandir o banco de dados (adicionando um tipo de registro ou item de dados), variar as restrições ou reduzir o banco de dados (removendo um tipo de registro ou item de dados). No último caso, os esquemas externos que se referem apenas aos dados remanescentes não precisariam ser afetados. Por exemplo, o esquema externo da Figura 1.4a não deveria ser afetado pela mudança do arquivo HISTORICO_ESCOLAR mostrado na Figura 1.2 para aquele exibido na Figura 1.5a. Somente a definição da visão e os mapeamentos precisam ser alterados em um SGBD que suporta a independência lógica dos dados. Após uma reorganização lógica no esquema conceitual, os programas que utilizam os construtores do esquema externo devem funcionar como antes da reorganização. As alterações nas restrições podem ser aplicadas ao esquema conceitual, sem afetar os esquemas externos ou os programas.
2. **Independência física de dados:** refere-se à capacidade de mudar o esquema interno sem ter de alterar o esquema conceitual. Consequentemente, o esquema externo também não precisa ser modificado. As mudanças no esquema interno podem ser necessárias para que alguns arquivos físicos possam ser reorganizados — por exemplo, pela criação de estruturas de acesso adicionais — para aperfeiçoar o desempenho da recuperação ou atualização de dados. Se os mesmos dados permanecem como anteriormente no banco de dados, não deveríamos ter de alterar o esquema conceitual. Por exemplo, criando uma rota de acesso (*access path*) para aumentar a velocidade de recuperação de registros de DISCIPLINAS (Figura 1.2) por Semestre e Ano, não será preciso que uma consulta como 'listar todas as disciplinas oferecidas no segundo semestre de 1998' fosse alterada, embora a consulta pudesse ser executada de forma mais eficiente pelo SGBD por meio do novo caminho de acesso.

Se temos um SGBD múltiplo-nível, seu catálogo deve ser expandido para incluir as informações de como solicitar o mapeamento e dados entre os vários níveis. O SGBD usa software adicional para realizar esses mapeamentos utilizando a informação de mapeamento no catálogo. A independência de dados ocorre porque quando o esquema é alterado em algum nível, o

esquema no próximo nível acima permanece sem mudanças; apenas o mapeamento entre os dois níveis é modificado. Por isso, os programas que se referem ao esquema do nível mais alto não precisam ser alterados.

A arquitetura de três-esquemas pode tornar mais fácil a independência de dados, tanto física quanto lógica. Entretanto, os dois níveis de mapeamentos criam uma sobrecarga (*overhead*) durante a compilação ou a execução de uma consulta ou de um programa, provocando ineficiências no SGBD. Por causa disso, poucos SGBDs têm implementadas toda a arquitetura de três-esquemas.

2.3 LINGUAGEM DE BANCO DE DADOS E INTERFACES

Na Seção 1.4 discutimos a variedade de usuários que utilizam o suporte de um SGBD. O SGBD deve oferecer linguagens e interfaces apropriadas para cada categoria de usuários. Nesta seção, abordaremos os tipos de linguagens e interfaces fornecidas por um SGBD e as categorias de usuários que são atendidas por interface.

2.3.1 Linguagens de SGBD

Como o projeto de um banco de dados está completo e um SGBD foi escolhido para implementá-lo, o primeiro ponto é especificar os esquemas conceitual e interno para o banco de dados e os mapeamentos entre os dois. Em muitos SGBDs, nos quais não existe uma separação específica de níveis, uma linguagem, chamada **linguagem de definição de dados** — *Data Definition Language* (DDL) —, é usada pelo Database Administrator (DBA) e pelos projetistas do banco de dados para definir ambos os esquemas. O SGBD terá um compilador DDL cuja função é processar os comandos DDL a fim de identificar os construtores e para armazenar a descrição do esquema no catálogo do SGBD.

Nos SGBDs, em que uma clara separação é mantida entre os níveis conceitual e interno, a DDL é usada para especificar somente o esquema conceitual. Outra linguagem, a **linguagem de definição de armazenamento** — *storage definition language* (SDL) —, é utilizada para especificar o esquema interno. Os mapeamentos entre os dois esquemas podem ser estabelecidos em qualquer uma dessas linguagens. Para uma verdadeira arquitetura de três-esquemas, necessitaríamos de uma terceira linguagem, a **linguagem de definição de visões** — *view definition language* (VDL) —, para especificar as visões dos usuários e os seus mapeamentos para o esquema conceitual, mas na maioria dos SGBDs, a DDL é usada para definir ambos os esquemas, o conceitual e o externo.

Quando os esquemas do banco de dados estiverem compilados e o banco de dados populado com os dados, os usuários devem ter alguns meios para manipular esse banco. As manipulações típicas são a recuperação, inserção, remoção e modificação dos dados. O SGBD fornece uma série de operações, ou uma linguagem chamada **linguagem de manipulação de dados** — *data manipulation language* (DML) —, com essa finalidade.

Nos SGBDs atuais, os tipos precedentes de linguagens são *consideradas linguagens não distintas*; ao contrário, uma linguagem integrada abrangente é usada e inclui construções para as definições do esquema conceitual, das visões e da manipulação de dados. A definição do armazenamento é mantida separada, pois é utilizada para obter a estrutura física de armazenamento que otimize a performance do sistema de banco de dados, atividade executada pela equipe do DBA. Um exemplo típico de linguagem de banco de dados abrangente é a linguagem relacional de banco de dados SQL (capítulos 8 e 9), que representa uma combinação da DDL, VDL e DML, como também os comandos para especificação de restrições, evolução de esquema e outros recursos. A SDL era um componente nas versões anteriores da SQL, porém foi removida da linguagem para mantê-la somente nos níveis conceitual e externo.

Há dois tipos principais de DMLs. A de **alto nível**, ou **não procedural**, pode ser usada para especificar suas próprias operações complexas no banco de dados de forma concisa. Muitos SGBDs permitem que os comandos DML de alto nível sejam introduzidos interativamente de um monitor ou terminal, ou embutidos em uma linguagem de programação de propósito geral. Nesse caso, os comandos DML devem ser identificados dentro do programa para que possam ser extraídos por um pré-compilador e processados pelo SGBD. A DML de **nível baixo**, ou **procedural**, *precisa* ser embutida em uma linguagem de programação de propósito geral. Esse tipo de DML recupera os registros individuais ou objetos do banco de dados e os processa separadamente. Por essa razão, é necessário usar os construtores de linguagens de programação, como o *loop*, para recuperar e processar cada registro de um conjunto de registros. As DMLs de nível baixo também são chamadas DMLs **um-registro-por-vez** (*record-at-a-time*) por causa dessa propriedade. As DMLs de alto nível, como a SQL, podem especificar e restaurar muitos registros em um único comando, por isso são chamadas DMLs **um-conjunto-por-vez** (*set-at-a-time*) ou **orientadas-a-conjunto** (*set-oriented*). Uma consulta em DML de alto nível frequentemente especifica *quais* dados recuperar em vez de *como* recuperá-los; como consequência, essas linguagens também são chamadas **declarativas**.

2.3 Linguagem de banco de dados e interfaces 25

Se os comandos DML de alto ou baixo nível forem embutidos em um programa de linguagem de propósito geral, a linguagem será chamada linguagem hospedeira (*host language*), e a DML, sublinguagem de dados. Porém, a DML de alto nível usada de maneira interativa e isoladamente é chamada linguagem de consulta (*query language*). Em geral, os comandos recuperar e atualizar da DML de alto nível podem ser usados interativamente e são, por isso, considerados parte da linguagem de consulta.

Os usuários finais casuais empregam normalmente uma linguagem de consulta de alto nível para especificar seus pedidos, considerando que os programadores usam a DML na sua forma embutida. Para os usuários iniciantes e parametrizáveis, existem interfaces amigáveis para interagir com o banco de dados que podem, também, ser utilizadas por usuários casuais ou outros que não queiram aprender sobre os detalhes de uma linguagem de consulta de alto nível. Discutiremos esses tipos de interfaces a seguir.

2.3.2 As Interfaces do SGBD

As interfaces amigáveis para os usuários fornecidas por um SGBD podem abranger o seguinte:

Interfaces Baseadas em Menus para os Clientes Web ou Navegação (*Browsing*). Essas interfaces são apresentadas ao usuário com listas de opções, chamadas menus, que o guiam durante a formulação de uma pesquisa. Os menus anulam a necessidade de memorizar comandos específicos e a sintaxe da linguagem de consulta; em vez disso, a consulta é composta passo a passo marcando-se as opções de um menu mostrado pelo sistema. Os menus suspensos (*pullDown*) são uma técnica muito popular nas interfaces para os usuários baseadas na Web. Esses menus são, geralmente, usados nas interfaces para navegação e permitem ao usuário pesquisar o conteúdo de um banco de dados de forma exploratória e não estruturada.

Interfaces Baseadas em Formulários. A interface baseada em formulário exibe um formulário para cada usuário. Os usuários podem preencher todos os campos do formulário de entrada para inserir um novo dado ou preencher somente alguns campos — nesse caso, o SGBD vai recuperar os dados faltantes para os campos vazios. Os formulários são normalmente projetados e programados para os usuários iniciantes como interfaces para transações customizadas. Muitos SGBDs têm linguagens para especificação de formulários, que são linguagens especiais que auxiliam os programadores a determinar os formulários. Alguns sistemas contêm funcionalidades que permitem que o usuário final construa, interativamente, um formulário na tela.

Interfaces Gráficas para os Usuários. Uma interface gráfica para o usuário — *graphical user interface* (GUI) — exibe um esquema para o usuário em um formulário diagramático. O usuário pode especificar uma consulta manipulando o diagrama. Em muitos casos, as GUIs utilizam ambos, os menus e os formulários. A maioria das GUIs usa um dispositivo indicador, como um mouse, para marcar certas partes do diagrama mostrado.

Interfaces de Linguagem Natural. Essas interfaces aceitam solicitações escritas em inglês ou em outros idiomas e tentam 'entendê-las'. Uma interface de linguagem natural normalmente tem seu próprio esquema, similar àquele conceitual do banco de dados, bem como um dicionário com as palavras importantes. A interface de linguagem natural refere-se às palavras em seu esquema, bem como a um conjunto de palavras-padrão no seu dicionário, para interpretar a solicitação. Se a interpretação for bem-sucedida, a interface gera uma consulta de alto nível correspondendo à solicitação da linguagem natural e a submete ao processamento pelo SGBD — e um diálogo é iniciado com o usuário para esclarecer a solicitação.

Interfaces para Usuários Parametrizáveis. Os usuários parametrizáveis, como os caixas de bancos, geralmente utilizam um pequeno conjunto de operações que são realizadas repetidamente. Os analistas de sistemas e os programadores projetam e implementam uma interface especial para cada classe conhecida de usuários. Em geral, uma pequena série de comandos adaptados é disponibilizada, com o objetivo de minimizar o número de teclas utilizadas para cada solicitação. Por exemplo, as teclas de funções em um terminal podem ser programadas para iniciar os vários comandos. Isso permite ao usuário parametrizável trabalhar com um número mínimo de teclas.

- 8 Em banco de dados orientados a objeto, a linguagem hospedeira e as sublinguagens de dados normalmente formam uma linguagem integrada: por exemplo, C++ com algumas extensões para suportar as funcionalidades de banco de dados. Alguns sistemas relacionais também possuem linguagens integradas: por exemplo, a PL/SQL do Oracle.
- 9 De acordo com o significado da palavra *query* (questão/consulta) em inglês, ela deveria ser usada para descrever somente as recuperações, e não as atualizações.

Interfaces **para o DBA**. A maioria dos sistemas de bancos de dados contém comandos privilegiados que podem ser usados somente pelos DBAs. Esses comandos incluem criação de contas, sistema de ajuste de parâmetros, autorização para criação de contas, mudança de esquemas e reorganização de estruturas de armazenamento do banco de dados.

2.4 O AMBIENTE DE SISTEMAS DE BANCO DE DADOS

Um SGBD é um complexo sistema de software. Nesta seção, discutiremos os tipos de componentes que constituem um SGBD e os tipos de sistemas com os quais o programa interage.

2.4.1 Módulos Componentes do SGBD

A Figura 2.3 ilustra, de forma simplificada, os componentes típicos de um SGBD. O banco de dados e o catálogo de SGBD são normalmente armazenados no disco. O acesso ao disco é controlado principalmente pelo **sistema operacional** (SO), que organiza as entradas e as saídas. Um módulo de **gerenciamento dos dados armazenados** de alto nível do SGBD controla o acesso à informação do SGBD que está armazenada no disco, se for parte do banco de dados ou do catálogo.

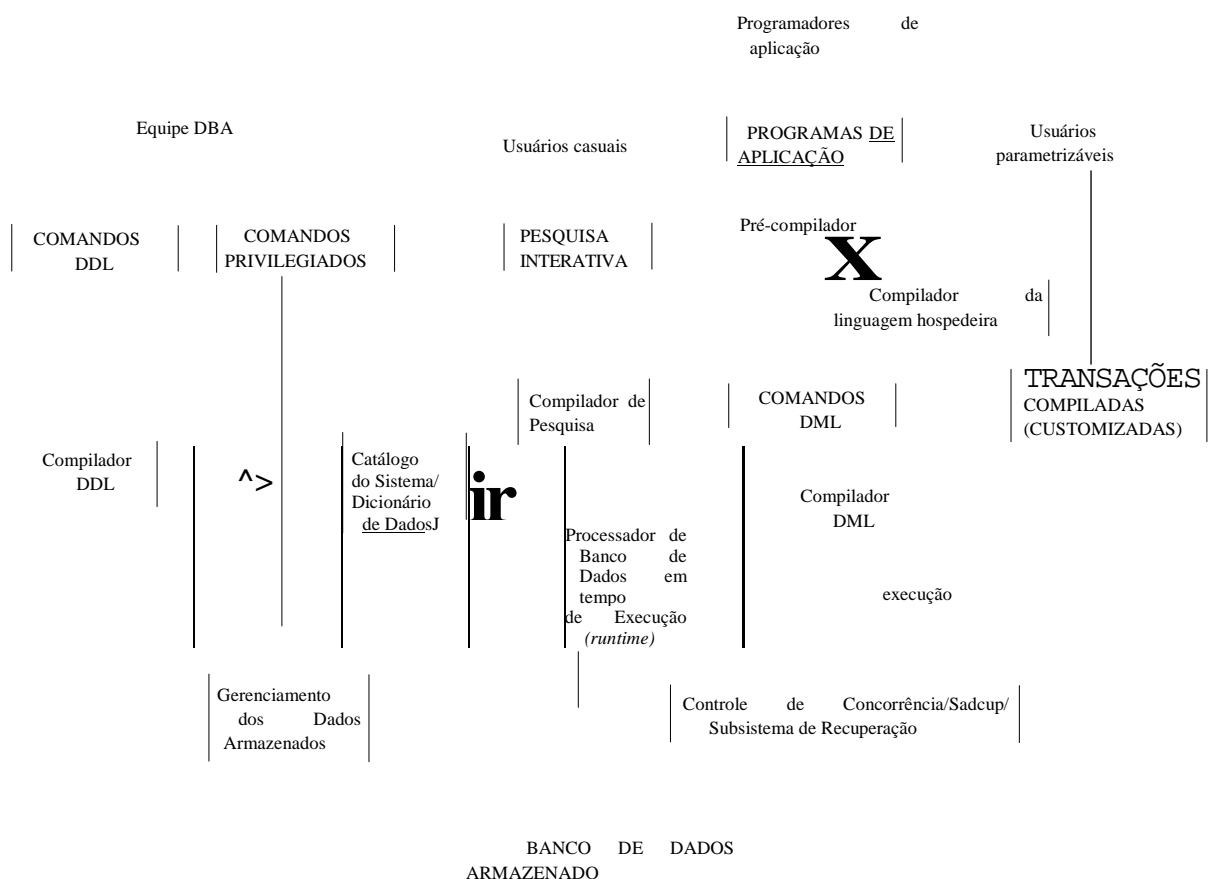


FIGURA 2.3 Módulos componentes de um SGBD e suas interações.

Os círculos com pontos pretos, identificados com as letras A, B, C, D e E, ilustram os acessos controlados pelo gerenciador de dados armazenados. O gerenciador pode usar os serviços básicos do SO para executar a transferência de dados entre o disco e a memória principal do computador, mas controla outros aspectos da transferência de dados, como a manipulação dos *buffers* na memória. Uma vez que o dado está nos *buffers* da memória principal, pode ser processado por outros módulos do SGBD, bem como pelos programas da aplicação. Alguns SGBDs têm seus próprios **módulos de gerenciamento de buffers**, enquanto outros usam o SO para manipular a buferização das páginas de disco.

O compilador DDL processa as definições do esquema, especificadas na DDL, e armazena as descrições dos esquemas (metadados) no catálogo do SGBD. O catálogo inclui informações como nomes e tamanhos dos arquivos, nomes e tipos de

itens de dados, detalhes de armazenamento de cada arquivo, informações de mapeamentos entre os esquemas e restrições, além de muitas outras informações necessárias para os módulos do SGBD. Os módulos de software do SGBD acessam as informações do catálogo conforme necessário.

O **processador de banco de dados em tempo de execução** (*runtime*) controla o acesso ao banco de dados em tempo de execução, recebe os comandos para a recuperação ou atualização e os executa no banco de dados. Os acessos passam pelo gerenciador de dados armazenados, e o gerenciador de *buffer* mantém as informações sobre as páginas do banco de dados na memória. O **compilador de consulta** (*query*) manipula as consultas de alto nível que são feitas interativamente. Ele analisa a sintaxe, compila ou interpreta a consulta criando um código de acesso ao banco de dados, e então gera as chamadas ao processador em tempo de execução para executar o código.

O **pré-compilador** extrai os comandos DML dos programas escritos em uma linguagem de programação hospedeira. Esses comandos são enviados para o compilador **DML** para compilação, gerando códigos para o acesso ao banco de dados. O restante do programa é enviado para o compilador da linguagem de programação hospedeira. Os códigos-objeto para os comandos DML e o restante do programa são acoplados, formando uma transação customizada cujo código executável inclui as chamadas para o processador em tempo de execução.

Atualmente é comum ter um **programa cliente** que acessa o SGBD de outro computador separado daquele em que está o banco de dados. O primeiro é chamado **computador cliente**, e o último, **servidor de banco de dados**. Em alguns casos, o cliente acessa um computador intermediário, o **servidor de aplicação**, que, por sua vez, acessa o servidor de banco de dados. Esse tópico será abordado na Seção 2.5.

A Figura 2.3 não pretende descrever um SGBD específico; ela ilustra os módulos típicos de um SGBD. O SGBD interage com o sistema operacional quando precisa acessar o disco — para o banco de dados ou para o catálogo. Se o sistema for compartilhado por vários usuários, o SO vai organizar a sequência de acessos do SGBD ao disco e o processamento do SGBD com os outros processos. Mas se o sistema de computador for dedicado principalmente a atuar como servidor de banco de dados, o SGBD vai controlar a buferização das páginas de disco na memória principal. O SGBD também interage com os compiladores de linguagens de programação hospedeiras genéricas, com os servidores de aplicação e com os programas clientes executados em máquinas separadas por meio da interface do sistema da rede.

2.4.2 Utilitários do Sistema de Banco de Dados

Além dos módulos de software descritos, muitos SGBDs têm **utilitários de banco de dados** que auxiliam o DBA no gerenciamento do sistema de banco de dados. Os utilitários comuns apresentam as seguintes funções:

- **Carregamento** (*loading*): esse utilitário é usado para carregar os arquivos de dados existentes — como os arquivos de texto ou os sequenciais — dentro do banco de dados. Normalmente, o formato corrente (fonte) do arquivo de dados e a estrutura do arquivo do banco de dados desejada (destino) são especificados para o utilitário, que então, automaticamente, formata os dados e os armazena no banco de dados. Com a proliferação dos SGBDs, a transferência de dados de um SGBD para outro tornou-se comum em muitas organizações. Alguns vendedores estão oferecendo produtos que geram programas apropriados de carregamento (*loading*) dadas a fonte existente e as descrições de armazenamento no banco de dados do destino (esquemas internos). Essas ferramentas também são ditas **ferramentas de conversão**.
- **Backup**: um utilitário de backup cria uma cópia do banco de dados, geralmente descarregando (*dumping*) todo o banco de dados em uma fita. A cópia back-up pode ser usada para restaurar o banco de dados, em caso de falhas catastróficas. Os backups incrementais também são usados se ocorrerem apenas alterações depois que um back-up prévio tenha sido gravado. O backup incremental é mais complexo, mas economiza espaço.
- **Reorganização de arquivos**: esse utilitário pode ser usado para reorganizar um arquivo de banco de dados em uma nova forma buscando melhorar seu desempenho.
- **Monitoramento de desempenho**: esse utilitário monitora o uso do banco de dados e fornece estatísticas para o DBA. O DBA utiliza essas estatísticas para tomar decisões, como reorganizar ou não os arquivos para melhorar o desempenho.

Outros utilitários podem estar disponíveis para classificação (*sorting*) de arquivos, execução de compressão de dados, monitoramento de acesso pelos usuários, interface com a rede e execução de outras funções.

2.4.3 Ferramentas, Ambientes de Aplicação e Funcionalidades de Comunicação

Existem, ainda, outras ferramentas disponíveis para os projetistas de bancos de dados, para os usuários e para os DBAs. As ferramentas CASE são usadas na fase de projeto do sistema de banco de dados. Outra ferramenta que pode ser muito útil em grandes organizações é o **sistema de dicionário de dados** (ou **repositório de dados**). Além de guardar informações sobre os esquemas e restrições no catálogo, o dicionário de dados armazena outras informações, como as decisões de projeto, os padrões de utilização, as descrições dos programas das aplicações e as informações dos usuários. Esse sistema também é chamado **repositório de informações**. Essa informação pode ser acessada *diretamente* pelos usuários ou pelo DBA quando necessário. Um dicionário de dados é similar a um catálogo do SGBD, mas inclui maior variedade de informações acessadas, principalmente pelos usuários, e não só pelo software do SGBD.

Os **ambientes de desenvolvimento de aplicação**, como o PowerBuilder (Sybase) ou o JBuilder (Borland), estão se tornando muito populares. Eles oferecem um ambiente para o desenvolvimento de aplicações de banco de dados, e incluem funcionalidades que auxiliam em muitas facetas do sistema de banco de dados como no projeto, desenvolvimento de GUIs, consulta e atualização, bem como desenvolvimento de programas de aplicação.

O SGBD também precisa de uma interface com o **software de comunicações**, cuja função é permitir aos usuários remotos o acesso ao banco de dados por meio de terminais de computador, estações de trabalho ou seus computadores pessoais. Estão conectados ao site do banco de dados por intermédio do hardware de comunicação de dados, como as linhas de telefone, redes locais, redes de longa distância ou dispositivos de comunicação via satélite. Muitos sistemas de bancos de dados comerciais possuem pacotes de comunicação que trabalham com o SGBD. Este, integrado ao pacote do sistema de comunicações, é denominado sistema DB/DC (*database/data Communications*). Além disso, alguns SGBDs distribuídos estão fisicamente difundidos em várias máquinas. Nesse caso, as redes de comunicação são necessárias para conectar as máquinas. Geralmente são **redes locais** — *local area networks* (LANs) —, no entanto, também podem ser outro tipo de rede.

2.5 ARQUITETURAS CENTRALIZADAS E CLIENTE/SERVIDOR PARA OS SGBDs

2.5.1 Arquitetura SGBD Centralizada

As arquiteturas dos SGBDs têm seguido tendências similares às aquelas de arquiteturas de sistemas de computadores. As primeiras arquiteturas utilizavam os grandes computadores centrais (*mainframes*) para processar todas as funções do sistema, incluindo os programas de aplicação e os de interface com os usuários, bem como todas as funcionalidades do SGBD. A razão era que a maioria dos usuários acessava o sistema via terminais de computador que não tinham poder de processamento e apenas ofereciam possibilidade de exibição. Então, todos os processos eram executados remotamente no sistema centralizado, e somente as informações de exibição e os controles eram enviados do computador para os terminais, conectados ao computador central por vários tipos de redes de comunicação.

Como os preços do hardware caíram, vários usuários mudaram seus terminais para os computadores pessoais (PCs) e estações de trabalho (*workstations*). Primeiro, os sistemas de banco de dados usavam esses computadores da mesma maneira que os terminais, então o SGBD ainda era um SGBD **centralizado**, no qual as funcionalidades, execuções de programas e processamento das interfaces com o usuário eram executados em uma única máquina. A Figura 2.4 ilustra os componentes físicos em uma arquitetura centralizada. Gradualmente, os sistemas SGBD começaram a explorar o poder de processamento disponível do lado do usuário, que direcionou as arquiteturas SGBD cliente-servidor.

2.5.2 Arquiteturas Cliente/Servidor Básicas

Primeiramente, discutiremos a arquitetura cliente-servidor genérica, depois, veremos como ela é aplicada ao SGBD. A **arquitetura cliente/servidor** foi desenvolvida para trabalhar com ambientes computacionais, nos quais um grande número de PCs, estações de trabalho, servidores de arquivo, impressoras, servidores de banco de dados, servidores Web e outros equipamentos estão conectados via rede. A idéia é definir os **servidores especializados** com funcionalidades específicas. Por exemplo, é possível conectar PCs ou pequenas estações de trabalho como clientes a um **servidor de arquivos** que mantém os arquivos das máquinas clientes.

10 Embora CASE signifique engenharia de software apoiada por computador (*computer-aided software engineering*), muitas ferramentas CASE são utilizadas, principalmente em projeto de banco de dados.

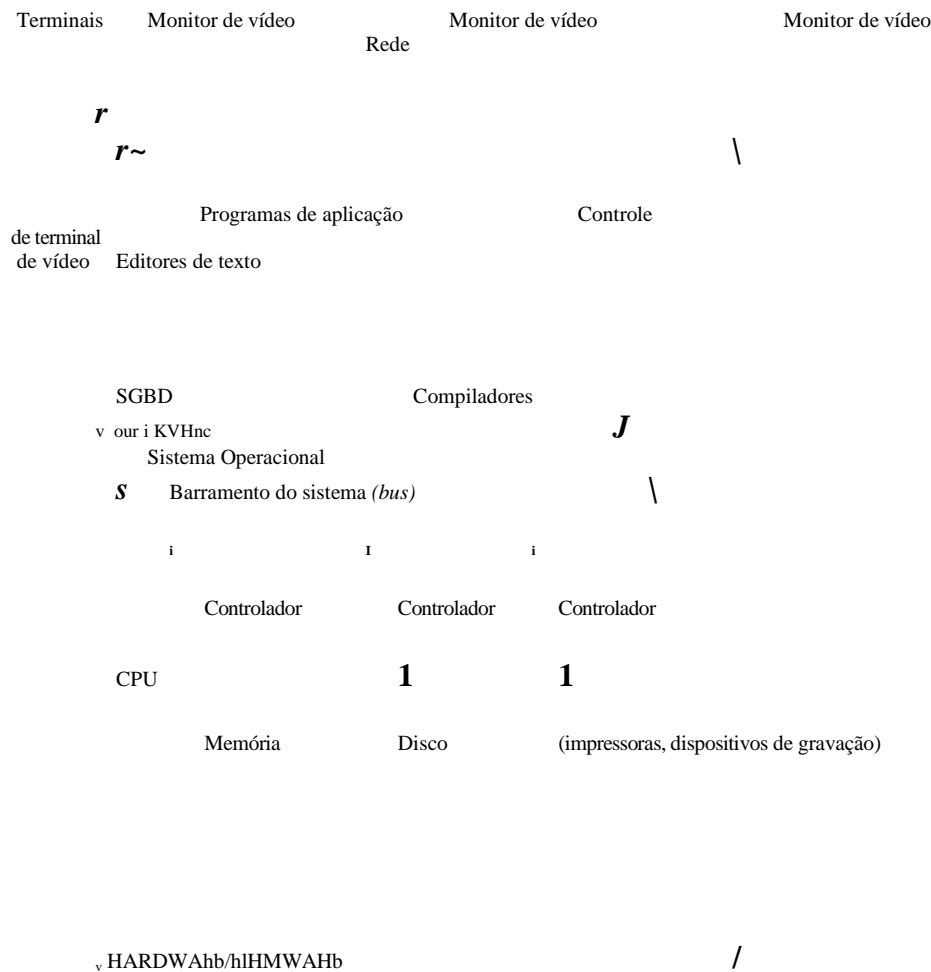


FIGURA 2.4 A arquitetura física centralizada.

Outra máquina poderia ser projetada como um **servidor de impressão**, sendo conectada a várias impressoras; conseqüentemente, todas as impressões solicitadas pelos clientes seriam encaminhadas para essa máquina. Os **servidores Web** ou **servidores de e-mails** também são especializados. Dessa forma, os recursos disponibilizados pelos servidores especializados podem ser acessados pelas diversas máquinas clientes. As **máquinas clientes** oferecem ao usuário as interfaces apropriadas para utilizar esses servidores, bem como o poder de processamento para executar as aplicações locais. Esse conceito pode ser transportado para o software, com o software especializado, como o SGBD, ou um pacote CAD (*computer aided design*) — projeto apoiado por computador —, armazenado nas máquinas servidores específicas e acessíveis a vários clientes. A Figura 2.5 ilustra a arquitetura de cliente/servidor no nível lógico, e a Figura 2.6 é um diagrama simplificado que mostra a arquitetura física. Algumas máquinas poderiam ser apenas sites clientes (por exemplo, estações de trabalho sem disco, estações de trabalho ou estações de trabalho/PCs com discos que tenham somente um software cliente instalado). Outras poderiam ser servidores dedicados. Outras máquinas poderiam, ainda, ter as funcionalidades de cliente e de servidor.

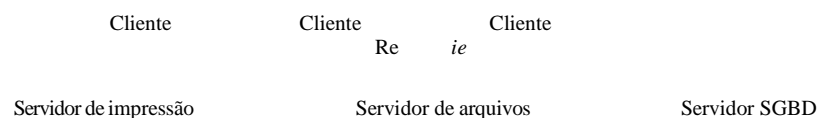


FIGURA 2.5 A arquitetura lógica de duas camadas cliente/servidor.

O conceito de arquitetura cliente/servidor possui uma estrutura fundamental que consiste em muitos PCs e estações de

trabalho, bem como um número pequeno de máquinas centrais (mainframes) conectadas via redes locais e outros tipos de redes de computadores. Um **cliente**, nessa estrutura, é em geral uma máquina de usuário que tem as funcionalidades de interface com o

usuário e processamento local. Quando um cliente precisa de uma funcionalidade adicional, como acesso ao banco de dados, inexistente naquela máquina, ela se conecta a um servidor que disponibiliza a funcionalidade. Um **servidor** é uma máquina que pode fornecer serviços para as máquinas clientes, como acesso a arquivos, impressão, arquivamento ou acesso a um banco de dados. Em geral, algumas máquinas instalam apenas o software cliente;

outras, somente o software servidor; e algumas podem incluir ambos, como pode ser visto na Figura 2.6. Entretanto, é mais comum que os softwares cliente e servidor

normalmente sejam executados em máquinas separadas. Dois tipos principais de arquiteturas de SGBD foram criados utilizando-se os fundamentos da estrutura cliente/servidor: duas e três camadas. Elas serão discutidas a seguir.

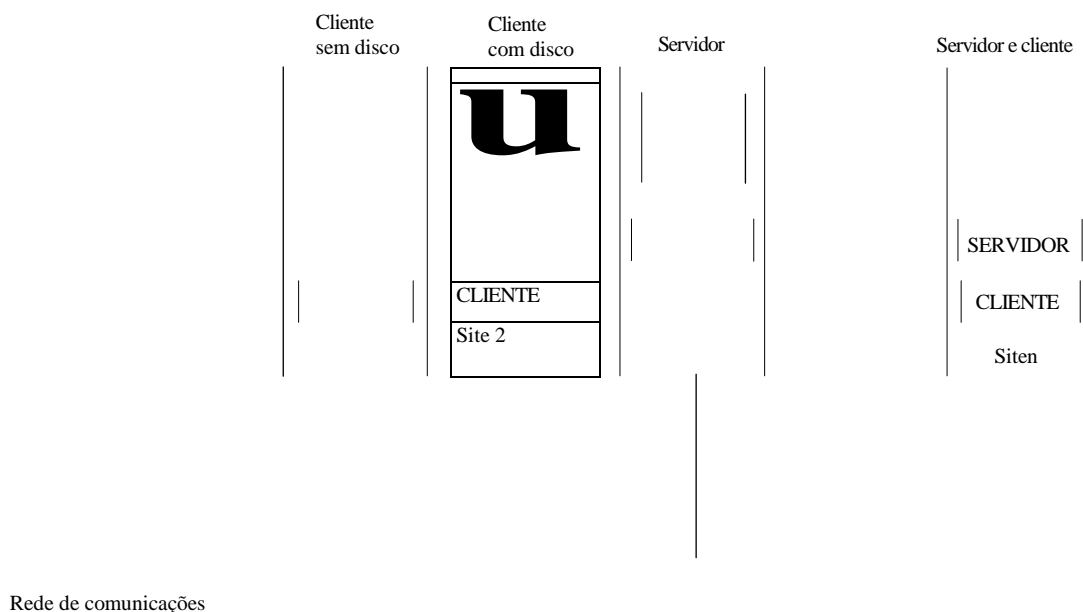


FIGURA 2.6 Arquitetura física cliente/servidor de duas camadas.

2.5.3 Arquiteturas Cliente/Servidor de Duas Camadas para os SGBDs

A arquitetura cliente/servidor está, cada vez mais, sendo incorporada aos pacotes comerciais de SGBD. No SGBD relacional (SGBDRs), muitos dos quais começaram como sistemas centralizados, os componentes de sistemas que primeiro foram movidos para o lado do cliente foram a interface com o usuário e os programas de aplicação. Pelo fato de a SQL (capítulos 8 e 9) oferecer uma linguagem-padrão para os SGBDRs, criou-se um ponto de divisão lógico entre o cliente e o servidor. Assim, a consulta e as funcionalidades de transação permaneceram do lado do servidor. Nessa arquitetura, o servidor geralmente é chamado **servidor de consulta** ou **servidor de transação**, porque fornece essas duas funcionalidades. Nos SGBDRs, o servidor é normalmente conhecido por **servidor SQL**, pois a maioria dos servidores SGBDR são baseados na linguagem e padrões SQL.

Em uma arquitetura cliente/servidor, os programas de interface com o usuário e os de aplicação podem ser executados no lado do cliente. Quando o acesso ao SGBD é necessário, o programa estabelece uma conexão com o programa (que está do lado do servidor); uma vez criada a conexão, o programa do cliente pode comunicar-se com o SGBD. Um padrão denominado **Open Database Connectivity (ODBC) — Conectividade a Banco de Dados Aberta** — fornece as **interfaces para o programa de aplicação** — (*application programming interface* ou API) —, o qual permite que os programas do lado do cliente se conectem ao SGBD, desde que ambas as máquinas cliente e servidor tenham o software necessário instalado. A maioria dos fornecedores de SGBD oferece os *drivers* ODBC para seus sistemas. Consequentemente, o programa cliente pode, na verdade, conectar-se a vários SGBDRs e enviar pedidos de consultas e transações usando o ODBC API, que são, então, processados no site do servidor. Os resultados da consulta são enviados de volta para o programa do cliente, que pode processá-los ou exibi-los conforme a necessidade. Um padrão relacionado à linguagem de programação Java, chamado JDBC, também está sendo definido. Isso permite que o programa Java cliente acesse o SGBD por meio de uma interface-padrão.

A segunda abordagem para a arquitetura cliente/servidor foi adotada por alguns SGBDs orientados a objeto. Como muitos

desses sistemas foram desenvolvidos na era da arquitetura cliente/servidor, a abordagem adotada foi dividir os módulos de software do SGBD entre cliente e servidor de modo mais integrado. Por exemplo, o **nível servidor** pode incluir a parte do software SGBD responsável por manipular o armazenamento de dados em páginas de discos, controle de concorrência local e recuperação, buferização e *caching* de páginas de disco e outras funções. O **nível cliente** pode executar a interface do usuário; as funções de dicionário de dados; as interações do SGBD com os compiladores das linguagens de programação; a otimização

11 Existem várias outras arquiteturas cliente/servidor. Abordaremos apenas as duas mais básicas aqui. No Capítulo 25 veremos outras arquiteturas cliente/servidor e as arquiteturas distribuídas.

global de consultas; o controle de concorrência e a recuperação por múltiplos servidores; a estruturação de objetos complexos dos dados nos *buffers* e outras funções. Nessa abordagem, a interação entre cliente/servidor é mais coesa, feita internamente pelos módulos do SGBD — alguns deles residem no cliente e outros no servidor — em vez de pelos usuários. A divisão exata das funcionalidades varia de sistema para sistema. Nessa arquitetura cliente/servidor, o servidor é chamado **servidor de dados**, porque fornece os dados nas páginas do disco para o cliente. Esses dados podem então ser estruturados em objetos para os programas cliente pelo próprio SGBD no lado cliente.

As arquiteturas descritas aqui são chamadas **arquiteturas de duas camadas**, pois os componentes de software são distribuídos em dois sistemas: cliente e servidor. As vantagens dessa arquitetura são a simplicidade e a compatibilidade com os sistemas existentes. A emergência da World Wide Web mudou os papéis dos clientes e servidores, direcionando-os para a arquitetura de três camadas.

2.5.4 Arquiteturas Cliente/Servidor de Três Camadas para Aplicações Web

Muitas aplicações para a Web usam uma arquitetura chamada **arquitetura de três camadas**, que possui uma camada intermediária entre o cliente e o servidor de banco de dados, como ilustrado na Figura 2.7. Essa camada intermediária, ou **camada do meio**, é, algumas vezes, chamada **servidor de aplicações** ou **servidor Web**, dependendo da aplicação. Esse servidor desempenha um papel intermediário armazenando as regras de negócio (procedimentos ou restrições) que são usadas para acessar os dados do servidor de banco de dados. Também pode incrementar a segurança do banco de dados checando as credenciais do cliente antes de enviar uma solicitação ao servidor de banco de dados. Os clientes possuem interfaces GUI e algumas regras de negócio adicionais específicas para a aplicação. O servidor intermediário aceita as solicitações do cliente, processa-as e envia comandos de banco de dados ao servidor de banco de dados, e então atua como um condutor por passar (parcialmente) os dados processados do servidor de banco de dados para os clientes — dados que podem ser processados novamente e filtrados para a apresentação aos usuários em um formato GUI. Desse modo, a *interface com o usuário*, as *regras de aplicação* e o *acesso aos dados* atuam como três camadas.

Os avanços na tecnologia de criptografia tornam mais segura a transferência dos dados sensíveis, criptografados, do servidor para o cliente, os quais serão descriptografados. A descriptografia pode ser feita por hardware ou por software avançados. Essa tecnologia oferece níveis elevados de segurança dos dados, mas as questões relativas à segurança da rede continuam preocupando os especialistas. Várias tecnologias para a compressão dos dados também estão auxiliando na transferência de grandes quantidades de dados dos servidores para os clientes nas redes com ou sem fios.

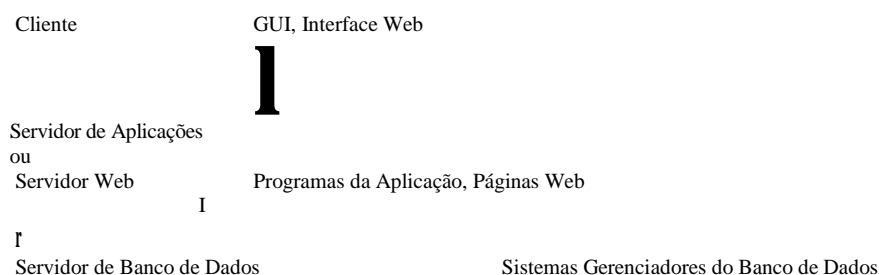


FIGURA 2.7 Arquitetura lógica cliente/servidor de três camadas.

2.6 CLASSIFICAÇÃO DOS SISTEMAS GERENCIADORES DE BANCO DE DADOS

Vários critérios são utilizados para a classificação dos SGBDs. O primeiro é o **modelo de dados** no qual o SGBD é baseado. O principal modelo de dados utilizado na maioria dos SGBDs comerciais atualmente é o **modelo de dados relacional**. O **modelo de dados de objeto** foi implementado em alguns sistemas comerciais, mas seu uso não foi muito difundido. Muitas aplicações legadas (antigas) ainda

utilizam os sistemas de banco de dados baseados nos **modelos hierárquicos** e de **rede**. Os SGBDs relacionais estão evoluindo continuamente e, em especial, têm incorporado muitos dos conceitos que foram desenvolvidos nos

bancos de dados de objetos. Essa evolução gerou uma nova categoria de SGBDs chamada SGBDs **objeto-relacional**. Podemos, portanto, classificar os SGBDs com base no modelo de dados: relacional, objeto, objeto-relacional, hierárquico, de rede e outros.

O segundo critério empregado para classificar os SGBDs é o **número de usuários** suportado pelo sistema. Os **sistemas de usuário único** suportam apenas um usuário por vez e são mais usados em computadores pessoais. Os **sistemas multiusuários**, que incluem a maioria dos SGBDs, auxiliam múltiplos usuários simultaneamente.

Um terceiro critério é o **número de sites** pelos quais o banco de dados está distribuído. Um SGBD é **centralizado** se o dado for armazenado em um único site. Um SGBD centralizado pode suportar múltiplos usuários, mas o SGBD e o banco de dados estão alocados em um único site. Um SGBD **distribuído** (SGBDD) pode ter o banco de dados e o software do SGBD distribuídos em vários sites conectados pela rede. Os SGBDDs homogêneos usam o mesmo software de SGBD em todos os sites. Uma tendência recente é desenvolver um software para acessar vários bancos de dados autônomos e preexistentes armazenados em SGBDs **heterogêneos**. Isso gera um SGBD **confederado** (ou **sistema de múltiplos bancos de dados**), no qual os SGBDs participantes são fracamente acoplados e têm um nível de autonomia local. Muitos SGBDDs utilizam a arquitetura cliente/servidor.

Um quarto critério é o **custo** do SGBD. A maioria dos pacotes de SGBD custa entre dez mil e cem mil dólares. Os sistemas de usuário único para a plataforma baixa, que trabalham com microcomputadores, custam entre cem dólares e três mil dólares. No outro lado da escala, poucos pacotes mais elaborados custam mais de cem mil dólares.

Podemos também classificar um SGBD baseado em opções de **tipos de caminhos de acesso** (*access path*) para os arquivos armazenados. Uma família de SGBD bem conhecida está baseada nas estruturas de arquivo invertidas. Finalmente, um SGBD pode ser utilizado para **propósitos gerais** ou para propósitos especiais. Quando o desempenho é considerado essencial, um SGBD específico pode ser projetado e construído para uma determinada aplicação não podendo ser usado para outras aplicações sem grandes alterações. Muitos sistemas para reservas de vôos e consultas a listas telefônicas, desenvolvidos no passado, são SGBDs com propósitos especiais. Eles pertencem à categoria de sistemas de **processamento de transações on-line** — *online transaction processing (OLTP)* —, os quais devem suportar um grande número de transações simultâneas sem gerar retardos excessivos.

Resumidamente, vamos descrever o principal critério para a classificação dos SGBDs: o modelo de dados. O modelo de dados relacional básico representa um banco de dados como uma coleção de tabelas, na qual cada uma delas pode ser armazenada como um arquivo separado. O banco de dados na Figura 1.2 é mostrado de maneira similar à da representação relacional. A maioria dos bancos de dados relacionais usa uma linguagem de consulta de alto nível chamada SQL e suporta de forma limitada as visões (*views*) de usuários. Discutiremos o modelo relacional, suas linguagens e operações, bem como técnicas para a programação de aplicações nos capítulos 5 a 9.

O modelo de dados de objetos define um banco de dados em termos de objetos, suas propriedades e operações. Os objetos com a mesma estrutura e comportamento pertencem a uma **classe**, e as classes são organizadas em **hierarquias** (*ou grafos acíclicos*). As operações de cada classe são especificadas nos termos de procedimentos pré-definidos, denominados **métodos**. Os SGBDs relacionais têm estendido seus modelos para incorporar os conceitos de banco de dados de objetos e outras funcionalidades. Esses sistemas são o **objeto relacional** ou o **sistema relacional estendido**, que serão vistos nos capítulos 20 a 22.

Dois modelos de dados antigos, importantes historicamente, agora conhecidos como modelos de dados legados, são os modelos de rede e os modelos hierárquicos. O **modelo de rede** representa os dados como tipos de registros e um tipo relacionamento 1:N, limitado, chamado **tipo conjunto**. A Figura 2.8 mostra um diagrama esquemático de rede para o banco de dados da Figura 1.2. Nele, os tipos registro são mostrados como retângulos e os tipos conjunto são vistos como setas diretas rotuladas. O modelo de rede, também conhecido como modelo CODASYL DBTG, tem uma linguagem um-registro-por-vez associada que precisa ser embutida em uma linguagem de programação hospedeira. O **modelo hierárquico** representa os dados como estruturas de árvores hierárquicas. Cada hierarquia representa um número de registros relacionados. Não há nenhuma linguagem-padrão para o modelo hierárquico, embora a maioria dos SGBDs hierárquicos possua linguagens um-registro-por-vez. Apresentaremos uma breve visão geral dos modelos de rede e hierárquicos nos apêndices E e F, no site deste livro.

O **modelo XML** (eXtended Markup Language), agora considerado o padrão para o intercâmbio de dados na Internet, também usa estruturas de árvores hierárquicas. Ele combina conceitos de banco de dados com os de modelos de representação de documentos. O dado é representado como elementos, que podem ser aninhados para criar estruturas hierárquicas complexas. Esse modelo assemelha-se conceitualmente àquele de objeto, mas usa uma terminologia diferente. Discutiremos o modelo XML e como ele é relacionado com os bancos de dados no Capítulo 26.

- 12 CODASYL DBTG significa *Conference on Data Systems Languages Data Base Task Group* (Conferência do Grupo Tarefa das Linguagens de Sistemas de Dados de Banco de Dados), que é o comitê que especifica o modelo de banco de dados de rede e

suas linguagens.

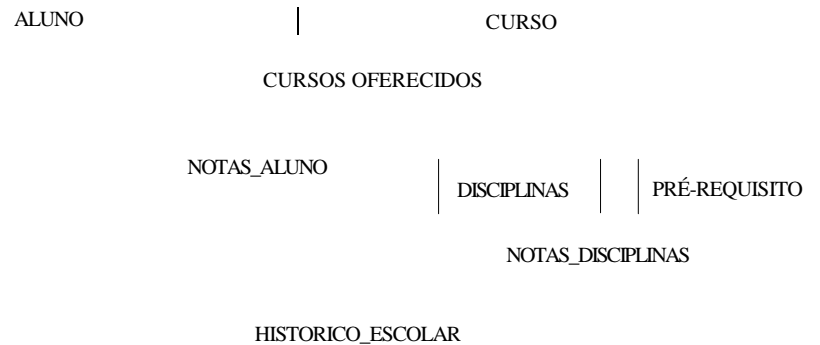


FIGURA 2.8 O esquema da Figura 2.1 em notação do modelo de rede.

2.7 RESUMO

Neste capítulo, demos uma introdução aos principais conceitos utilizados nos sistemas de banco de dados. Definimos um modelo de dado e destacamos as três principais categorias dos modelos de dados:

- Alto nível ou modelos de dados conceituais (baseados em entidades e relacionamentos).
- Baixo nível ou modelos de dados físicos.
- Modelos de dados representacionais ou de implementação (baseado-em-registro, orientado a objeto).

Diferenciamos o esquema, ou descrição de um banco de dados, do banco de dados efetivo. O esquema não é alterado freqüentemente, já o estado do banco de dados é alterado toda vez que o dado é inserido, apagado ou modificado. Descrevemos a seguir a arquitetura de três-esquemas do SGBD, que permite três níveis de esquemas:

- Um esquema interno descreve a estrutura de armazenamento físico do banco de dados.
- Um esquema conceitual é uma descrição de alto nível de todo o banco de dados.
- Os esquemas externos descrevem as visões de diferentes grupos de usuários.

Um SGBD que separa claramente os três níveis deve ter mapeamentos entre os esquemas para transformar as solicitações e os resultados de um nível para o próximo. A maioria dos SGBDs não separa os três níveis completamente. Usamos a arquitetura de três-esquemas para definir os conceitos de independência de dados lógica e física.

Em seguida, abordamos os principais tipos de linguagens e interfaces que os SGBDs suportam. Uma linguagem de definição de dados (DDL) é usada para definir o esquema conceitual do banco de dados. Na maioria dos SGBDs, a DDL também define as visões dos usuários, e algumas vezes as estruturas de armazenamento; em outros SGBDs, podem existir outras linguagens (VDL, SDL) para especificação das visões e estruturas de armazenamento. O SGBD compila todas as definições de esquema e armazena suas descrições no seu catálogo. Uma linguagem de manipulação de dados (DML) é empregada para especificar as recuperações e atualizações do banco de dados. As DMLs podem ser de alto nível (orientadas-a-conjunto, não procedural) ou de baixo nível (orientadas-a-registro, procedural). Uma DML de alto nível pode ser embutida em uma linguagem de programação hospedeira, ou pode ser utilizada sozinha (*stand-alone*) — nesse caso, é chamada linguagem de consulta (*query*).

Discutimos os diferentes tipos de interfaces fornecidas pelo SGBD e os tipos de usuários de SGBD aos quais cada interface está associada. A seguir, discutimos o ambiente do sistema de banco de dados, os módulos típicos de software do SGBD e as funcionalidades do programa para auxiliar os usuários e o DBA a executar suas tarefas. Depois, demos uma visão geral das arquiteturas de duas e três camadas para as aplicações de banco de dados, que são atualmente muito comuns na maioria das aplicações modernas, particularmente nas aplicações de banco de dados para a Web.

Na seção final, classificamos os SGBDs de acordo com alguns critérios: modelo de dados, número de usuários, número de sites, custo, tipos de caminhos de acesso e generalidade. A principal classificação dos SGBDs está baseada no modelo de dados. Resumidamente, vimos os principais modelos de dados usados nos SGBDs comerciais atuais.

Questões de Revisão

2.1 Defina os seguintes termos: *modelo de dados*, *esquema de banco de dados*, *estado do banco de dados*, *esquema interno*, *esquema conceitual*, *esquema externo*, *independência de dados*, *DDL*, *DML*, *SDL*, *VDL*, *linguagem de consulta*, *linguagem hospedeira*, *sublinguagem de dados*, *funcionalidade do banco de dados*, *catálogo*, *arquitetura cliente/servidor*.

- 2.2 Discuta as principais categorias dos modelos de dados.
- 2.3 Qual a diferença entre um esquema de banco de dados e um estado de banco de dados?
- 2.4 Descreva a arquitetura de três-esquemas. Por que precisamos de mapeamentos entre os níveis dos esquemas? Como diferentes linguagens de definição de esquemas suportam essa arquitetura?
- 2.5 Qual a diferença entre independência de dados lógica e física?
- 2.6 Qual a diferença entre as DMLs procedural e não procedural?
- 2.7 Discuta os diferentes tipos de interfaces amigáveis para o usuário (GUIs) e os tipos de usuário típico de cada uma.
- 2.8 Defina com que outro tipo de sistema de software de computador um SGBD interage?
- 2.9 Qual a diferença entre as arquiteturas cliente/servidor de duas e três camadas?
- 2.10 Discuta alguns tipos de funcionalidades de banco de dados, ferramentas e suas funções.

Exercícios

- 2.11 Pense nos diferentes usuários para o banco de dados da Figura 1.2. Que tipos de aplicações cada usuário necessitaria? Para qual categoria de usuário cada um pertenceria, e de que tipo de interface precisaria?
- 2.12 Escolha uma aplicação de banco de dados com a qual você está familiarizado. Projete um esquema e mostre um banco de dados exemplo para essa aplicação, usando a notação das figuras 2.1 e 1.2. Que tipo de informação adicional e restrições você gostaria de representar no esquema? Pense nos vários usuários para o seu banco de dados e projete uma visão para cada um.

Bibliografia Seleccionada

Muitos livros de banco de dados, entre os quais os dos autores Date (2001), Silberschatz *et al.* (2001), Ramakrishnan e Gehrke (2002), Garcia-Molina *et al.* (1999, 2001) e Abiteboul *et al.* (1995), possuem uma discussão dos vários conceitos de banco de dados apresentados aqui. Tsichritzis e Lochovsky (1982) é o precursor dos modelos de dados. Tsichritzis e Klug (1978) e Jar-dine (1977) apresentam a arquitetura de três-esquemas, que foi, primeiramente, sugerida no relatório DBTG CODASYL (1971) e, mais tarde, em um relatório do American National Standards Institute (Ansi, 1975). Uma análise profunda do modelo de dados relacional, com suas possíveis extensões, é feita em Codd (1992). O padrão proposto para os bancos de dados orientados a objeto é descrito em Cattell (1997). Muitos documentos descrevendo o XML estão disponíveis na Web, como o XML (2003).

Os exemplos de funcionalidades de bancos de dados estão no ETI Extract Toolkit (www.eti.com), e a ferramenta para a administração de bancos de dados está em DB Artisan from Embarcadero Technologies (www.embarcadero.com).

3

Modelagem de Dados Usando o Modelo Entidade- Relacionamento

A modelagem conceitual é uma fase muito importante no planejamento de uma aplicação de um banco de dados bem-sucedida. Geralmente, o termo **aplicação de um banco de dados** refere-se a um banco de dados particular e aos programas a ele associados, que implementam consultas e atualizações. Por exemplo, uma aplicação de um banco de dados BANCO, para manter as contas dos clientes, conteria programas para implementar as atualizações correspondentes aos depósitos e retiradas deles. Esses programas oferecem interfaces gráficas — *User-Friendly Graphical User Interfaces* (GUIs) —, que utilizam formulários e menus para interagir com os usuários finais da aplicação — os caixas do banco, nesse exemplo. Por isso, a parte da aplicação de um banco de dados exigirá o projeto, implementação e testes desses **programas de aplicação**. Tradicionalmente, o projeto e teste dos programas de aplicação são tratados mais no domínio da engenharia de software do que no de um banco de dados. Como as metodologias de projeto de banco de dados priorizam os conceitos de operações em objetos de banco de dados, e as metodologias de engenharia de software tratam mais especificamente os detalhes da estrutura dos bancos de dados que os programas de software vão usar e acessar, está claro que essas atividades estão fortemente relacionadas. Discutiremos brevemente alguns dos conceitos para as operações em um banco de dados no Capítulo 4 e, novamente, a metodologia de projeto de um banco de dados, com exemplos de aplicações, no Capítulo 12 deste livro.

Neste capítulo, seguiremos a abordagem tradicional: concentração nas estruturas e restrições do banco de dados durante seu projeto. Apresentaremos os conceitos de modelagem do **modelo Entidade-Relacionamento (ER)**, que é um modelo de dados conceitual de alto nível, além de muito popular. Esse modelo e suas variações são normalmente empregados para o projeto conceitual de aplicações de um banco de dados, e muitas ferramentas de projeto de um banco de dados aplicam seus conceitos. Descreveremos os conceitos da estruturação de dados básica e as restrições do modelo ER, e discutiremos seu uso no projeto de esquemas conceituais para aplicações de bancos de dados. Apresentaremos também a notação diagramática associada ao modelo ER, conhecida por **diagramas ER**.

As metodologias de modelagem de objetos como **UML** (*Universal Modeling Language* — **Linguagem de Modelagem Universal**) estão se tornando cada vez mais populares no projeto e engenharia de software. Essas metodologias vão além do projeto de um banco de dados, especificando o projeto detalhado dos módulos de software e suas interações, utilizando vários tipos de diagramas. Uma importante parte dessas metodologias — os *diagramas de classe* — é similar, sob muitos aspectos, aos diagramas ER. As *operações* em objetos são definidas nos diagramas de classe que, além disso, especificam a estrutura do esquema do banco de dados. As operações podem ser usadas para especificar os *requisitos funcionais* durante o projeto do banco de dados, conforme discutido na Seção 3.1. Na Seção 3.8 apresentaremos algumas notações e conceitos da UML para os diagramas de classe, que são particularmente relevantes para o projeto de um banco de dados, e os comparamos, brevemente, às notações e conceitos de ER. As notações e os conceitos adicionais da UML serão apresentados na Seção 4.6 e no Capítulo 12.

- 1 Uma classe é similar, sob muitos aspectos, a um *tipo entidade*.

Este capítulo está organizado como segue. A Seção 3.1 discute o papel dos modelos de dados de alto nível conceitual no projeto de um banco de dados. Na Seção 3.2 introduziremos os requisitos para uma aplicação de um banco de dados exemplo, a fim de ilustrar o uso dos conceitos do modelo ER. Esse banco de dados exemplo também é usado nos capítulos subsequentes. Na Seção 3.3 apresentaremos os conceitos de entidades e atributos e, gradualmente, a técnica diagramática para exibir a representação de um esquema ER. Na Seção 3.4 vamos abordar os conceitos de relacionamentos binários, seus papéis e restrições estruturais. Na Seção 3.5, os tipos entidade fraca. A Seção 3.6 mostrará como um esquema é refinado para incluir os relacionamentos. A Seção 3.7 vai rever a notação para diagramas ER, resumindo as decisões de projeto e discutindo como escolher os nomes para a construção do esquema de um banco de dados. A Seção 3.8 introduzirá alguns conceitos de diagramas de classe UML, comparando-os aos do modelo ER e aplicando-os ao mesmo banco de dados exemplo. A Seção 3.9 resume o capítulo.

O material na Seção 3.8 não é obrigatório para um curso introdutório, devendo ser estudado se desejado. Porém, para uma cobertura mais completa dos conceitos de modelagem de dados e de projeto de um banco de dados conceitual, o leitor deverá conhecer o material do Capítulo 4, após concluir o Capítulo 3. O Capítulo 4 descreve as extensões do modelo ER que levam ao modelo ER-Estendido (EER), que inclui conceitos como especialização, generalização, herança e tipos de união (categorias). No Capítulo 4 introduzimos também alguns conceitos adicionais e a notação UML.

3.1 USANDO MODELOS DE DADOS DE ALTO NÍVEL CONCEITUAL PARA O PROJETO DE UM BANCO DE DADOS

A Figura 3.1 mostra uma descrição simplificada do processo de projeto de banco de dados. O primeiro passo é o **levantamento e análise de requisitos**. Durante essa etapa, o projetista entrevista o possível usuário do banco de dados para entender e documentar seus **requisitos de dados**. O resultado dessa etapa é o registro conciso dos requisitos do usuário. Esses requisitos deveriam ser especificados em um formulário, da forma mais detalhada e completa possível. Em paralelo à especificação dos requisitos de dados, é útil definir os requisitos **funcionais** conhecidos da aplicação. Esses requisitos consistem em **operações** (ou **transações**) definidas pelo usuário que serão empregadas no banco de dados, incluindo as recuperações e atualizações. No projeto de software, é comum o uso de *diagramas de fluxo de dados*, *diagramas de sequência*, *cenários* e outras técnicas para a especificação de requisitos funcionais. Não discutiremos nenhuma dessas técnicas aqui porque elas são, geralmente, descritas com detalhes em textos de engenharia de software. Daremos uma visão de algumas dessas técnicas no Capítulo 12.

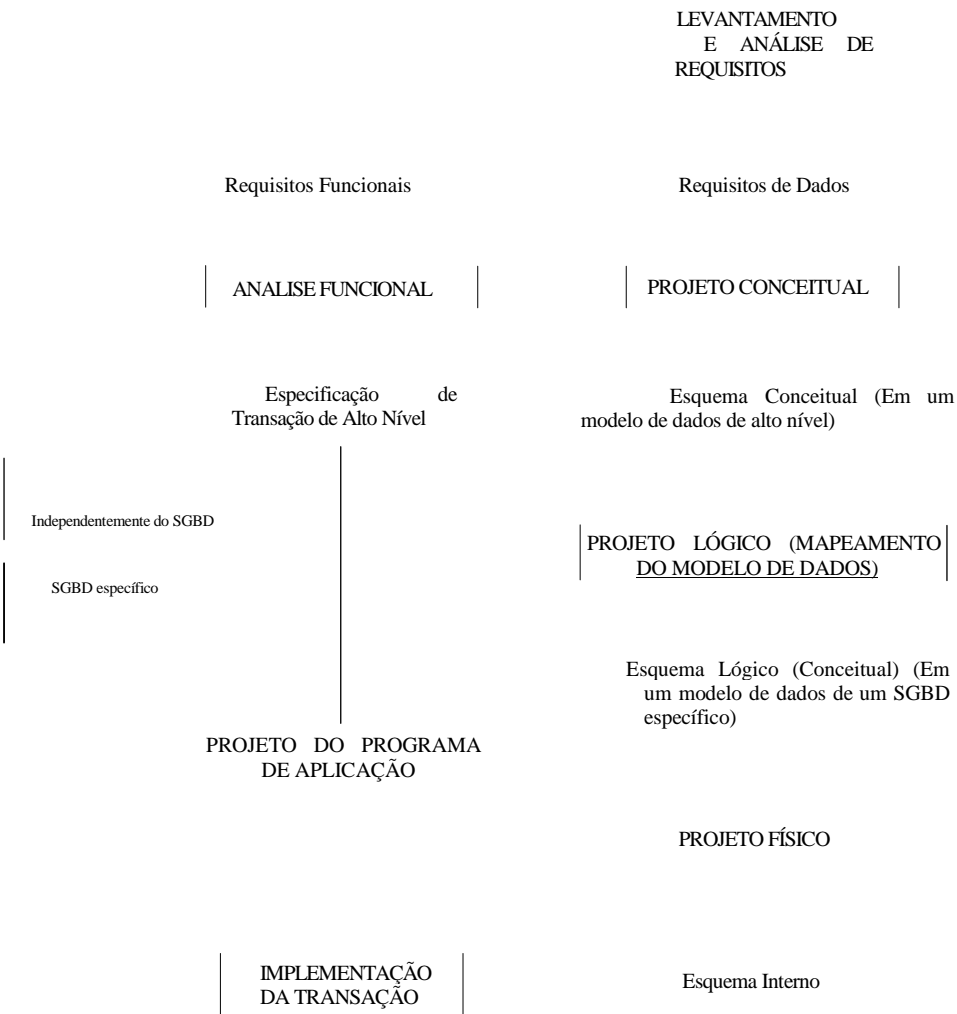
Uma vez que todos os requisitos tenham sido levantados e analisados, o próximo passo é criar um **esquema conceitual** para o banco de dados, utilizando um modelo de dados conceitual de alto nível. Essa fase é chamada **projeto conceitual**. O esquema conceitual é uma descrição concisa dos requisitos de dados dos usuários e inclui descrições detalhadas de tipos entidade, relacionamentos e restrições — são expressos usando os conceitos fornecidos pelo modelo de dados de alto nível. Como esses conceitos não incluem detalhes de implementação, eles são, normalmente, mais fáceis de entender e podem ser empregados na comunicação com os usuários não-técnicos. O esquema conceitual de alto nível também pode ser usado como uma referência para assegurar que todos os requisitos de dados do usuário sejam atendidos e não entrem em conflito. Essa abordagem permite que os projetistas de banco de dados se concentrem na especificação das propriedades do dado, sem se preocupar com os detalhes de armazenamento. Conseqüentemente, é mais fácil apresentarem um bom projeto conceitual do banco de dados.

Durante ou após o projeto do esquema conceitual, as operações básicas do modelo de dados podem ser usadas para especificar as operações de alto nível do usuário, identificadas durante a análise funcional. Servem também para confirmar que o esquema conceitual reúne todos os requisitos funcionais identificados. As modificações no esquema conceitual podem ser feitas se alguns requisitos funcionais não puderem ser especificados usando-se o esquema inicial.

A próxima etapa no projeto do banco de dados é a implementação real do banco de dados utilizando um SGBD comercial. A maioria dos SGBDs comerciais atuais usa um modelo de dados de implementação — como o relacional ou o modelo de banco de dados objeto-relacional — de forma que o esquema conceitual seja transformado de um modelo de dados de alto nível em um modelo de dados de implementação. Essa fase é conhecida como **projeto lógico** ou **mapeamento do modelo de dados**, e o seu resultado é um esquema do banco de dados no modelo de dados de implementação do SGBD.

O último passo é a fase do projeto físico, durante a qual são definidas as estruturas de armazenamento interno, índices, caminhos de acesso e organizações de arquivo para os arquivos do banco de dados. Em paralelo a essas atividades são projetados e implementados os programas de aplicação, como transações do banco de dados correspondentes às especificações de transação de alto nível. No Capítulo 12 discutiremos o processo do projeto de um banco de dados mais detalhadamente.

Neste capítulo apresentaremos apenas os conceitos básicos do modelo ER para o projeto do esquema conceitual. Os conceitos adicionais de modelagem são discutidos no Capítulo 4, quando introduziremos o modelo EER.



Programas de Aplicação **FIGURA 3.1** Um diagrama simplificado para ilustrar as principais fases do projeto de um banco de dados.

3.2 UMA APLICAÇÃO EXEMPLO DE BANCO DE DADOS

Nesta seção apresentamos um exemplo de uma aplicação de banco de dados chamado EMPRESA, que serve para ilustrar os conceitos básicos do modelo ER e seu uso no projeto do esquema. Listamos os requisitos para o banco de dados, então criamos o esquema conceitual passo a passo, enquanto introduzimos os conceitos do modelo ER. O banco de dados EMPRESA controla os

empregados da empresa, os departamentos e os projetos. Vamos supor que, após a fase de levantamento e análise dos requisitos, o projetista tenha fornecido a seguinte descrição do 'minimundo' — parte da empresa a ser representada no banco de dados:

1. A empresa está organizada em departamentos. Cada departamento tem um nome único, um número único e um empregado que gerencia o departamento. Temos a data em que o empregado começou a gerenciar o departamento. E este pode ter diversas localizações.
2. Um departamento controla um número qualquer de projetos, cada qual com um único nome, um único número e uma única localização.
3. Armazenamos o nome de cada empregado, o número do seguro social, endereço, salário, sexo e data de nascimento. Um empregado está alocado a um departamento, mas pode trabalhar em diversos projetos que não são controlados, necessariamente, pelo mesmo departamento. Controlamos o número de horas semanais que um empregado trabalha em cada projeto. Também controlamos o supervisor direto de cada empregado.
4. Queremos ter o controle dos dependentes de cada empregado para fins de seguro. Guardamos o primeiro nome, sexo, data de nascimento de cada dependente, e o parentesco dele com o empregado.

A Figura 3.2 mostra como o esquema para essa aplicação de um banco de dados pode ser representado por meio de notações gráficas — os diagramas ER. Descrevemos, passo a passo, o processo de derivação desse esquema a partir dos requisitos declarados, e explicaremos a notação diagramática do ER enquanto introduzimos, na seção seguinte, os conceitos do modelo ER.

FIGURA 3.2 Um diagrama do esquema ER para o banco de dados EMPRESA.

Número do seguro social, ou SSN, é um identificador único de nove dígitos, atribuído a cada indivíduo nos Estados Unidos, para controlar seu emprego, benefícios e taxas. Outros países têm, normalmente, esquemas similares de identificação, como números de cartão de identificação pessoal. No Brasil, o CPF é similar ao SSN, entretanto, com função diferente e menos abrangente; manteremos o SSN nos exemplos utilizados. (N.deT.)

3.3 TIPOS ENTIDADE, CONJUNTOS DE ENTIDADE E ATRIBUTOS-CHAVE

O modelo ER descreve os dados como *entidades*, *relacionamentos* e *atributos*. Na Seção 3.3.1 apresentamos os conceitos de entidades e seus atributos. Na Seção 3.3.2 discutimos os tipos entidade e os atributos-chave. E na Seção 3.3.3 definimos o projeto conceitual inicial dos tipos entidade para o banco de dados EMPRESA. Na Seção 3.4 são descritos os relacionamentos.

3.3.1 Entidades e Atributos

Entidades e Seus Atributos. O objeto básico que o modelo ER representa é uma entidade, 'algo' do mundo real, com uma existência independente. Uma entidade pode ser um objeto com uma existência física (por exemplo, uma pessoa, um carro, uma casa ou um funcionário) ou um objeto com uma existência conceitual (por exemplo, uma empresa, um trabalho ou um curso universitário).

Cada entidade tem atributos — propriedades particulares que a descrevem. Por exemplo, uma entidade empregado pode ser descrita pelo nome do empregado, idade, endereço, salário e trabalho (função). Uma dada entidade terá um valor para cada um de seus atributos. Os valores dos atributos que descrevem cada entidade se tornarão a maior parte dos dados armazenados no banco de dados.

A Figura 3.3 mostra duas entidades e os valores de seus atributos. A entidade empregado ej tem quatro atributos: Nome, Endereço, Idade e FoneResidencial; seus valores são 'John Smith', '2311 Kirby, Houston, Texas 77001', '55' e '713-749-2630', respectivamente. A entidade empresa q possui três atributos: Nome, Sede e Presidente. Seus valores são 'Sunco Oil', 'Houston' e 'John Smith', respectivamente.

Nome = John Smith	Nome = Sunco Oil
Endereço = 2311 Kirby, Houston, Texas 77001	Sede = Houston
Idade = 55	
FoneResidencial = 713-749-2630	— Presidente = John Smith

FIGURA 3.3 Duas entidades, empregado e, e empresa c., e seus atributos.

Diversos tipos de atributos ocorrem no modelo ER: *simples* versus *composto*, *univalorado* versus *multivalorado*, e *armazenado* versus *derivado*. Primeiro, definiremos esses tipos de atributos e os ilustraremos por meio de exemplos. Depois, introduziremos o conceito de *valor null* (nulo) para um atributo.

Atributos Compostos versus Simples (Atômicos). Os atributos compostos podem ser divididos em subpartes menores, que representam a maioria dos atributos básicos com significados independentes. Por exemplo, o atributo Endereço da entidade empregado, mostrado na Figura 3.3, pode ser subdividido em EnderecoRua, Cidade, Estado e CEP, com os valores '2311 Kirby', 'Houston', 'Texas' e '77001'. Os atributos que não são divisíveis são chamados simples ou atributos atômicos. E os atributos compostos podem formar uma hierarquia; por exemplo, EnderecoRua pode ser subdividido, ainda, em três atributos simples: Rua, Número e Apartamento, conforme mostrado na Figura 3.4. O valor de um atributo composto é a concatenação dos valores componentes dos seus atributos simples.

Os atributos compostos são úteis para modelar as situações nas quais o usuário algumas vezes se refere ao atributo como um grupo e, em outras ocasiões, se refere especificamente a um de seus componentes. Se o atributo composto é referenciado apenas como um todo, não há necessidade de subdividi-lo em atributos componentes. Por exemplo, se não é necessário referir-se a um componente individual de um endereço (CEP, rua, e assim por diante), então o endereço pode ser definido como um atributo simples.

3 O *zip code* é o nome usado nos Estados Unidos para um código postal de cinco dígitos. *
Corresponde ao CEP no Brasil, portanto, utilizaremos CEP para designá-lo. (N. de T.)

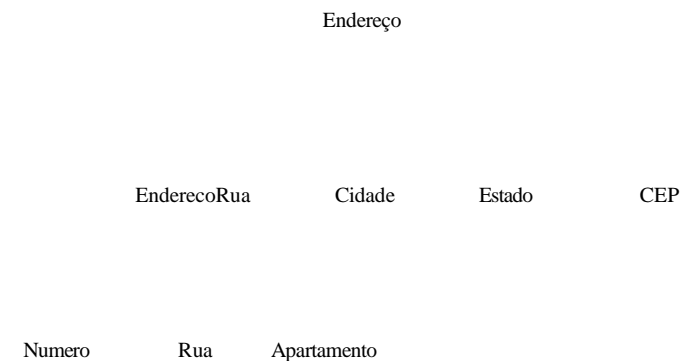


FIGURA 3.4 Uma hierarquia de atributos compostos.

Atributos Monovalorados *versus* Multivalorados. A maioria dos atributos tem um valor único para uma dada entidade; esses atributos são chamados monovalorados. Por exemplo, Idade é um atributo monovalorado de uma pessoa. Em alguns casos, um atributo pode ter um conjunto de valores para a mesma entidade — por exemplo, um atributo Cor para um carro ou um atributo Titulação para uma pessoa. Os carros com uma cor têm um valor único, enquanto aqueles com dois tons contêm dois valores para Cor. Da mesma forma, uma pessoa pode não ter um título acadêmico, outra pessoa pode ter um e, uma terceira pessoa, dois ou mais títulos, portanto, pessoas diferentes podem ter *números de valores* diferentes para o atributo Titulação. Esses atributos são chamados multivalorados. Um atributo multivalorado deve ter limite inferior e superior para restringir o número de valores permitidos a cada entidade individual. Por exemplo, o atributo Cor de um carro pode ter entre um e três valores, se presumirmos que um carro possa ter, no máximo, três cores.

Atributos Armazenados *versus* Derivados. Em alguns casos, dois (ou mais) valores de atributos estão relacionados — por exemplo, os atributos Idade e DataNascimento de uma pessoa. Para uma entidade pessoa em particular, o valor de Idade pode ser determinado pela data corrente (hoje) e o valor de DataNascimento da pessoa. Portanto, o atributo Idade é chamado atributo derivado, e é dito derivado do atributo DataNascimento, que, por sua vez, é chamado atributo armazenado. Alguns valores de atributos podem ser derivados de *entidades relacionadas*; por exemplo, um atributo NumeroDeEmpregados, de uma entidade departamento, pode ser derivado da contagem do número de empregados relacionados (que trabalham) nesse departamento.

Valores *nulls* (Nulos). Em alguns casos, determinada entidade pode não ter um valor aplicável a um atributo. Por exemplo, o atributo Apartamento de um endereço se aplica apenas a endereços que estão em edifícios de apartamentos, e não a outros tipos de residência, como as casas. Por analogia, um atributo Titulação só se aplica a pessoas com titulação acadêmica. Para essas situações é criado um valor especial chamado *null* (nulo). Um endereço de uma casa teria valor *null* para seu atributo Apartamento, e uma pessoa sem titulação acadêmica teria valor *null* para Titulação. O valor *null* pode ser usado também quando não conhecemos o valor de um atributo para uma entidade em particular; por exemplo, se não soubermos o telefone residencial de 'John Smith', na Figura 3.3. O sentido do primeiro tipo de *null* corresponde a *não aplicável*, ao passo que o sentido do último se refere a *desconhecido*. A categoria 'desconhecido' de *null* pode ser, ainda, classificada em dois casos. O primeiro aparece quando se sabe que o valor do atributo existe mas está *faltando* — por exemplo, se o atributo Altura de uma pessoa é dado como *null*. O segundo caso surge quando *não se sabe* se o valor do atributo existe — por exemplo, se o atributo FoneResidencial de uma pessoa é *null*.

Atributos Complexos. Observa-se que os atributos compostos e multivalorados podem ser aninhados de uma maneira arbitrária. Podemos representar essa organização arbitrária agrupando os componentes de um atributo composto entre parênteses (), separando os componentes por meio de vírgulas e mostrando os atributos multivalorados entre chaves {}. Esses atributos são chamados atributos complexos. Por exemplo, se uma pessoa pode ter mais de uma residência e cada uma delas pode ter múltiplos telefones, um atributo EnderecoFone para uma pessoa pode ser especificado como mostrado na Figura 3.5.

* O conceito de valor *null* corresponde à ausência de valor em português; seria correto, portanto, o uso do termo nulo. Entretanto, com o propósito de evitar as freqüentes confusões entre nulo e zero, manteremos, prioritariamente, o termo em inglês. (N. de T.)

4 Para aqueles familiarizados com XML observaríamos, aqui, que os atributos complexos são similares aos elementos complexos no XML (Capítulo 26).

3.3 Tipos Entidade, Conjuntos de Entidade e Atributos-chave 41

```
{EnderecoFone({Fone(CodigoArea, NumeroFone)}),
Endereco(EnderecoRua(Numero, Rua, Apartamento),
Cidade, Estado, CEP))}
```

FIGURA 3.5 Um atributo complexo: EnderecoFone.**3.3.2 Tipos Entidade, Conjuntos de Entidades e Conjuntos de Valores***

Tipos Entidade e Conjuntos de Entidade. Um banco de dados contém, geralmente, grupos de entidades que são similares. Por exemplo, uma empresa com centenas de empregados pode querer armazenar informações similares referentes a cada um deles. Essas entidades empregado compartilham os mesmos atributos, mas cada uma delas tem *seu(s) próprio(s) valor(es)* para cada atributo. Um tipo entidade define uma *coleção* (ou *conjunto*) de entidades que possuem os mesmos atributos. Cada tipo entidade no banco de dados é descrito por seu nome e atributos. A Figura 3.6 mostra dois tipos entidade, chamados EMPREGADO e EMPRESA, com uma lista de atributos para cada um. Algumas entidades individuais de cada tipo também são apresentadas com os valores de seus atributos. A coleção de todas as entidades de um tipo entidade em particular, em um ponto no banco de dados, é chamada conjunto de entidades, que normalmente é chamado pelo mesmo nome do tipo entidade. Por exemplo, EMPREGADO refere-se ao tipo *entidade*, assim como ao atual *conjunto de todas as entidades empregado* no banco de dados.

Um tipo entidade é representado nos diagramas ER (Figura 3.2) como uma caixa retangular, incluindo seu nome. Os nomes dos atributos são colocados em elipses e conectados ao seu tipo entidade por linhas retas. Os atributos compostos são conectados aos seus atributos componentes por linhas retas. Os atributos multivalorados são identificados por elipses duplas.

Um tipo entidade descreve o esquema ou conotação para um *conjunto de entidades* que possuem a mesma estrutura. O conjunto de entidades de um tipo entidade particular, agrupado em um conjunto, também é chamado extensão do tipo entidade.

Atributos-chave de um Tipo Entidade. Uma restrição importante das entidades de um tipo entidade é a chave ou restrição de unicidade em atributos. Um tipo entidade tem, geralmente, um atributo cujos valores são distintos para cada uma das entidades do conjunto de entidade. Esse atributo é chamado atributo-chave e seus valores podem ser usados para identificar cada entidade univocamente.

NOME DO TIPO ENTIDADE:	EMPREGADO	EMPRESA
-------------------------------	------------------	----------------

Nome, Idade, Salário	Nome, Sede Administrativa, Presidente
----------------------	---------------------------------------

CONJUNTO DE ENTIDADE: (EXTENSÃO)

FIGURA 3.6 Dois tipos entidade, EMPREGADO e EMPRESA, e algumas entidades-membro de cada um.

Por exemplo, na Figura 3.6, o atributo Nome é uma chave do tipo entidade EMPRESA, porque não é permitido que duas empresas tenham o mesmo nome. Para o tipo entidade PESSOA, um atributo-chave típico é o NumeroSeguroSocial (SSN).

* Utilizamos os termos tipo relacionamento, tipo entidade em vez de tipo *de* relacionamento, tipo *de* entidade. Entendemos que o termo se refere à representação de diferentes abstrações e não a tipos *diferentes* de relacionamentos e entidades, daí a supressão do *de*. Dependendo do contexto, usaremos também os termos entidade ou relacionamento para nos referir a esses tipos. (N. de T.)

5 Estamos usando a notação de diagrama ER próxima à proposta original (Chen, 1976). Infelizmente, muitas outras notações são utilizadas atualmente. Ilustramos algumas delas no Apêndice A e no final deste capítulo, quando apresentaremos o diagrama de classe da UML.

(John Smith, 55, 80k)

•*.

(Fred Brown, 40, 30K)

*3. (Judy Clark, 25, 20K)

r

>

(Sunco Oil, Houston, John Smith) (Fast Computer, Dallas, Bob King)

42 Capítulo 3 Modelagem de Dados Usando o Modelo Entidade-Relacionamento

Algumas vezes, diversos atributos juntos formam uma chave, significando que a *combinação* dos valores dos atributos deve ser distinta para cada entidade. Se um conjunto de atributos possuir essa propriedade, a forma adequada de representação no modelo ER, que utilizaremos, será a definição de um *atributo composto* e sua designação como atributo-chave do tipo entidade. Observemos que essa chave composta deve ser *mínima*, isto é, todos os atributos componentes devem ser incluídos no atributo composto para garantir a propriedade da unicidade. Na notação diagramática ER, cada atributo-chave tem seu nome sublinhado dentro da elipse, conforme ilustrado na Figura 3.2.

Definir um atributo como chave de um tipo entidade significa que a propriedade de unicidade deve ocorrer para *toda conjunto de entidades* do tipo entidade. Portanto, essa restrição proíbe quaisquer duas entidades de terem, ao mesmo tempo, o mesmo valor para o atributo-chave. Essa não é a propriedade de uma extensão em particular — sem dúvida, é uma restrição de todas as *extensões* do tipo entidade. Essa restrição de chave (e outras restrições que discutiremos adiante) é derivada das restrições do minimundo que o banco de dados representa.

Alguns tipos entidade têm *mais de um* atributo-chave. Por exemplo, cada um dos atributos IDVeículo e Registro do tipo entidade CARRO (Figura 3.7) é uma chave por si só. O atributo Registro é um exemplo de uma chave composta, formada por dois atributos componentes simples, o NumeroRegistro e Estado — nenhum deles, sozinho, é uma chave. Um tipo entidade pode também *não ter chave*; nesse caso é chamado tipo *entidade fraca* (Seção 3.5).

CARRO

Registro(NumeroRegistro, Estado), IDVeículo, Marca, Modelo, Ano, {Cor}

car-1 • ((ABC 123, TEXAS), TK629, Ford Mustang, conversível, 1998, {vermelho, preto})

car-2 • ((ABC 123, NOVA YORK), WP9872, Nissan Máxima, 4 portas, 1999, {azul})

car-3 • ((VSY 720, TEXAS), TD729, Chrysler LeBaron, 4 portas, 1995, {branco, azul})

FIGURA 3.7 O tipo entidade CARRO com dois atributos-chave, Registro e IDVeículo.

Conjuntos de Valores (Domínios) de Atributos. Cada atributo simples de um tipo entidade está associado a um conjunto de valor (ou domínio de valores), que determina o conjunto de valores válidos para os atributos de cada entidade. Na Figura 3.6, se o limite de idade permitido para os empregados está entre 16 e 70 anos, podemos estabelecer que o conjunto de valores para o atributo Idade, de EMPREGADO, seja o conjunto de números inteiros entre 16 e 70. Por analogia, podemos determinar o conjunto de valores para o atributo Nome como o conjunto de *strings* dos caracteres alfabéticos, separados por caracteres brancos, e assim por diante. Os conjuntos de valores não são mostrados nos diagramas ER. Eles são determinados utilizando-se os tipos de dados básicos, disponíveis na maioria das linguagens de programação, como número inteiro, cadeia de caracteres (strings), booleano, número real com ponto flutuante, tipo enumerado, subfaixa, e assim por diante. Os tipos adicionais de dados para representar data, hora e outros conceitos também são empregados. Matematicamente, um atributo A de um tipo entidade E, cujo conjunto de valor é V, pode ser definido como uma função de E para o *power set* P(V) de V:

$A : E \rightarrow P(V)$

Referimo-nos ao valor do atributo A para a entidade e como $A(e)$. A definição prévia cobre ambos os atributos, os monovalorados e os multivalorados, bem como os nulos. Um valor nulo é representado por um conjunto *vazio*. Para atributos monovalorados, $A(e)$ está limitado a ser um *singleton set* para cada entidade e em E, enquanto em atributos multivalorados não

6 Os atributos supérfluos não devem ser incluídos em uma chave, entretanto, uma superchave pode contê-los, como será explanado no Capítulo 5.

7 O power set P(V) é um conjunto de subconjuntos de V.

8 Um *singleton set* é um conjunto com apenas um elemento (valor).

há restrição. Para um atributo composto A , o conjunto de valor V é o produto cartesiano de $P(V_1), P(V_2), \dots, P(V_n)$, em que V_1, V_2, \dots, V_n são os conjuntos dos valores componentes dos atributos simples que formam A .

$$V = P(V_1) \times P(V_2) \times \dots \times P(V_n)$$

3.3.3 Projeto Conceitual Inicial do Banco de Dados EMPRESA

Agora podemos definir os tipos entidade para o banco de dados EMPRESA com base nos requisitos descritos na Seção 3.2. Depois de definirmos os diversos tipos entidade e seus atributos, na Seção 3.4 *refinaremos* nosso projeto após introduzirmos o conceito de relacionamento. Conforme os requisitos apresentados na Seção 3.2 podemos identificar quatro tipos entidade — correspondentes a cada um dos quatro itens da especificação (Figura 3.8):

1. Um tipo entidade DEPARTAMENTO, com os atributos Nome, Numero, Localizações, Gerente e DataInicioGerencia. O atributo Localizações é o único multivalorado. Nome e Numero são atributos-chave (separados), pois foram definidos como únicos (não existe mais de um com o mesmo valor).
2. Um tipo entidade PROJETO, com os atributos Nome, Numero, Localização e DepartamentoControle. Nome e Numero são atributos-chave (separados).
3. Um tipo entidade EMPREGADO, com os atributos Nome, SSN (para o número do seguro social), Sexo, Endereço, Salário, DataNascimento, Departamento e Supervisor. Nome e Endereço podem ser atributos compostos, no entanto, isso não foi especificado nos requisitos. Devemos voltar aos usuários para ver se algum deles vai utilizar os componentes individuais de Nome — PrimeiroNome, InicialMeio, UltimoNome — ou de Endereço.
4. Um tipo entidade DEPENDENTE, com os atributos Empregado, NomeDependente, Sexo, DataNascimento e Parentesco (com o empregado).

DEPARTAMENTO Nome, Numero{Localizacoes}, Gerente,
DataInicioGerencia

PROJETO Nome, Numero, Localização,
DepartamentoControle

EMPREGADO
Nome (PNome, InicialIM, UNome), SSN, Sexo, Endereço, Salário,
DataNascimento, Departamento, Supervisor, {TrabalhaEm (Projeto, Horas)}

DEPENDENTE Empregado, NomeDependente, Sexo, DataNascimento,
Parentesco

FIGURA 3.8 Projeto inicial dos tipos entidade para o banco de dados EMPRESA.

Até o momento, não tivemos de representar o fato de um empregado poder trabalhar em diversos projetos ou o número de horas semanais que um empregado trabalha em cada projeto. Essa característica está listada como parte do requisito 3, na Seção 3.2, e pode ser representada por um atributo composto multivalorado de EMPREGADO chamado TrabalhaEm, com os componentes simples (Projeto, Horas). Como alternativa, pode ser representada por um atributo composto multivalorado de PROJETO conhecido como Trabalhadores, com os componentes simples (Empregado, Horas). Escolhemos a primeira alternativa, na Figura 3.8, que mostra os tipos entidade descritos. O atributo Nome, de EMPREGADO, é mostrado como um atributo composto, provavelmente decidido após consulta aos usuários.

3.4 TIPOS RELACIONAMENTO, CONJUNTOS DE RELACIONAMENTOS, PAPÉIS E RESTRIÇÕES ESTRUTURAIS

Na Figura 3.8 há diversos *relacionamentos implícitos* entre os vários tipos entidade. Quando um atributo de uma entidade refere-se a outra entidade há um relacionamento. Por exemplo, o atributo Gerente, de DEPARTAMENTO, corresponde a um empregado que gerencia o departamento; o atributo DepartamentoControle, de PROJETO, diz respeito ao departamento que controla o projeto; o atributo Supervisor, de EMPREGADO, refere-se a outro empregado (aquele que supervisiona esse empregado); o atributo Departamento, de EMPREGADO, corresponde ao departamento no qual o empregado trabalha, e assim por diante. No modelo ER, essas referências não deveriam ser representadas por atributos, mas por relacionamentos, que serão discutidos nesta seção. O

esquema do banco de dados EMPRESA será refinado na Seção 3.6 para representar, explicitamente, os relacionamentos. No projeto inicial dos tipos entidade, os relacionamentos são capturados na forma de atributos. A medida que o projeto é refinado, esses atributos são convertidos em relacionamentos entre as entidades.

A seguir, apresentamos a organização desta seção. A Seção 3.4.1 introduz os conceitos de tipos relacionamento e instâncias de relacionamento. Na Seção 3.4.2 definimos os conceitos de grau de relacionamento, nomes de papéis e relacionamentos recursivos. Na Seção 3.4.3 discutimos as restrições estruturais em relacionamentos — como graus de cardinalidade e dependências existentes. A Seção 3.4.4 mostra como os tipos relacionamento podem, também, ter atributos.

3.4.1 Tipos Relacionamento, Conjuntos e Instâncias

Um **tipo relacionamento** R entre n tipos entidade E_1, E_2, \dots, E_n define um conjunto de associações — ou um **conjunto de relacionamentos** — entre essas entidades. Como utilizado em tipos entidade e conjuntos de entidades, um tipo relacionamento e seu conjunto de relacionamentos correspondentes são, habitualmente, referidos pelo *mesmo nome*, R . Matematicamente, o conjunto de relacionamentos R é um conjunto de **instâncias** de relacionamento r_i , em que cada r_i associa-se a n entidades individuais (e_1, e_2, \dots, e_n), e cada entidade e^j em r_i é um membro do tipo entidade E_j , $1 < j \leq n$. Portanto, um tipo relacionamento é uma relação matemática em $E_1 \times E_2 \times \dots \times E_n$; como alternativa, ele pode ser definido como um subconjunto do produto cartesiano $E_1 \times E_2 \times \dots \times E_n$. Cada um dos tipos entidade E_1, E_2, \dots, E_n é dito **participante** do tipo relacionamento R ; analogamente, cada uma das entidades individuais e_1, e_2, \dots, e_n é conhecida como participante da instância de relacionamento $r_i = (e_1, e_2, \dots, e_n)$.

Informalmente, cada instância de relacionamento r_i em R é uma associação de entidades, na qual a associação inclui, exatamente, uma entidade de cada tipo entidade participante. Cada instância de relacionamento r_i representa o fato de as entidades participantes em r_i estarem relacionadas, de alguma forma, à situação do minimundo correspondente. Por exemplo, consideremos um tipo relacionamento TRABALHA_PARA, entre os dois tipos entidade, EMPREGADO e DEPARTAMENTO, que associa cada empregado ao departamento para o qual ele trabalha. Cada instância de relacionamento no conjunto de relacionamento TRABALHA_PARA associa uma entidade empregado a uma entidade departamento. A Figura 3.9 ilustra esse exemplo, em que é mostrada cada instância de relacionamento r_i conectada às entidades empregado e departamento participantes em r_i . No minimundo representado na Figura 3.9, os empregados e_1, e_3 e e_6 trabalham para o departamento d_1 ; e_2 e e_4 trabalham para d_2 ; e e_5 e e_7 trabalham para d_3 .

Nos diagramas ER, os tipos relacionamento são mostrados como caixas em forma de losango, que são conectadas por linhas retas às caixas retangulares, que representam as entidades participantes. O nome do relacionamento é mostrado na caixa em forma de losango (Figura 3.2).

TRABALHA_PARA

FIGURA 3.9 Algumas instâncias do conjunto de relacionamento TRABALHA_PARA, que representa um tipo relacionamento TRABALHA_PARA entre EMPREGADO e DEPARTAMENTO.

3.4.2 Grau de Relacionamento, Nomes de Papéis e Relacionamentos Recursivos

Grau de um Tipo Relacionamento. O grau de um tipo relacionamento é o número de entidades que participam desse relacionamento. Assim, o relacionamento TRABALHA_PARA é de grau dois. Um tipo relacionamento de grau dois é chamado binário, e um de grau três, ternário. Um exemplo de um relacionamento ternário é FORNECE, mostrado na Figura 3.10, em que cada instância de relacionamento r_i associa-se a três entidades — um fornecedor s , uma peça p e um projeto j — sempre que s fornece peça p ao projeto j . Normalmente, os relacionamentos podem ser de qualquer grau, mas os mais comuns são os relacionamentos binários. Os relacionamentos de grau mais alto são, normalmente, mais complexos que os binários; mais à frente, na Seção 4-7, os caracterizaremos.

Relacionamentos como Atributos. Às vezes, é conveniente pensarmos em um tipo relacionamento em termos de atributos, conforme apresentado na Seção 3.3.3. Consideremos o tipo relacionamento TRABALHA_PARA da Figura 3.9. Pode-se pensar em um atributo chamado Departamento, do tipo entidade EMPREGADO, cujo valor para cada entidade empregado é (uma referência a) a *entidade departamento* para o qual o empregado trabalha. Por isso, o conjunto de valores para esse atributo Departamento é o *conjunto de todas as entidades* DEPARTAMENTO, que é o conjunto de entidades DEPARTAMENTO. Isso é o que fizemos na Figura 3.8, quando especificamos o projeto inicial do tipo entidade EMPREGADO para o banco de dados EMPRESA. Entretanto, quando pensamos em um relacionamento binário como um atributo, sempre temos duas opções. Nesse exemplo, a alternativa é pensar em um atributo Empregados, multivalorado, do tipo entidade DEPARTAMENTO, cujos valores para cada entidade departamento seja o *conjunto de entidades empregado* que trabalha para o departamento. O conjunto de valores para esse atributo Empregados é o *power set* do conjunto de entidade EMPREGADO. Qualquer um desses dois atributos — Departamento de EMPREGADO ou Empregados de DEPARTAMENTO — pode representar o tipo relacionamento TRABALHA_PARA. Se ambos estão representados, eles são, obrigatoriamente, o contrário um do outro.

FORNECEDOR

FORNECE

FIGURA 3.10 Algumas instâncias de relacionamento do conjunto de relacionamento ternário FORNECE.

Nomes de Papéis e Relacionamentos Recursivos. Cada tipo entidade que participa de um tipo relacionamento executa um papel particular no relacionamento. O nome do papel significa o papel que uma entidade participante de um tipo entidade executa em cada instância de relacionamento, e ajuda a explicar o significado do relacionamento. Por exemplo, no tipo relacionamento TRABALHA_PARA, EMPREGADO executa o papel de *empregado* ou *trabalhador*, e DEPARTAMENTO desempenha o papel de *departamento* ou *empregador*.

⁹ Esse conceito de representação de tipos relacionamento como atributos é usado em uma classe de modelos de dados chamada modelos funcionais de dados. Em bancos de dados orientados a objeto (Capítulo 20), os relacionamentos podem ser representados por atributos de referência, cada qual em uma direção, ou em ambas as direções, inversamente. Em bancos de dados relacionais (Capítulo 5), chaves estrangeiras são um tipo de atributo de referência, usadas para representar os relacionamentos.

46 Capítulo 3 Modelagem de Dados Usando o Modelo Entidade-Relacionamento

Os nomes de papéis não são tecnicamente necessários em tipos relacionamento em que todos os tipos entidade participantes são distintos, uma vez que cada nome de tipo entidade participante pode ser usado como o nome do papel. Entretanto, em alguns casos, o *mesmo* tipo entidade participa mais de uma vez em um tipo relacionamento em *papéis diferentes*. Nesses casos, o nome do papel torna-se essencial para definir o sentido de cada participação. Esses tipos relacionamento são chamados relacionamentos recursivos. A Figura 3.11 apresenta um exemplo. O tipo relacionamento SUPERVISÃO relaciona um empregado a um supervisor, no qual ambas as entidades, empregado e supervisor, são membros do mesmo tipo entidade EMPREGADO. Assim, o tipo entidade EMPREGADO *participa duas vezes* em SUPERVISÃO: uma, no papel de *supervisor* (ou *chefe*), e outra, no papel de *supervisionado* (ou *subordinado*). Cada instância de relacionamento r_i em SUPERVISÃO associa duas entidades empregado, e^i e e_k , uma das quais executa o papel de supervisor, e a outra, o papel de supervisionado. Na Figura 3.11, as linhas marcadas com '1' representam o papel do supervisor, e as com '2', o papel do supervisionado. Assim, e_1 supervisiona e_2 e e_3 , e_4 supervisiona e_6 e e_7 , e e_5 supervisiona e_j e e^i .

3.4.3 Restrições em Tipos Relacionamento

Os tipos relacionamento têm, geralmente, certas restrições que limitam a possibilidade de combinações de entidades que podem participar do conjunto de relacionamentos correspondente. Essas restrições são determinadas pela situação do mini-mundo que os relacionamentos representam. Por exemplo, na Figura 3.9, se a empresa definiu uma regra em que cada empregado tem de trabalhar para exatamente um departamento, gostaríamos, então, de descrever essa restrição no esquema. Podemos distinguir dois tipos principais de restrições: *razão de cardinalidade* e *participação*.

EMPREGADO

SUPERVISÃO

FIGURA 3.11 Um relacionamento recursivo SUPERVISÃO entre EMPREGADO, no papel de *supervisora*, e EMPREGADO, no papel de *subordinado* (2).

Razões de Cardinalidade para Relacionamentos Binários. A razão de cardinalidade para um relacionamento binário especifica o número *máximo* de instâncias de relacionamento em que uma entidade pode participar. Por exemplo, no tipo relacionamento binário TRABALHA_PARA, DEPARTAMENTO:EMPREGADO tem razão de cardinalidade 1:N, significando que cada departamento pode estar relacionado a (isto é, emprega) qualquer número de empregados, mas um empregado pode estar relacionado a (trabalha para) apenas um departamento. As razões de cardinalidade possíveis para os tipos relacionamento binário são 1:1, 1:N, N:1 e M:N.

Um exemplo de um relacionamento binário de 1:1 é GERENCIA (Figura 3.12), que relaciona uma entidade departamento ao empregado que gerencia esse departamento. Isso representa as restrições do minimundo, onde — em um dado

10 N representa *qualquer número* de entidades relacionadas (zero ou mais).

3.4 Tipos Relacionamento, Conjuntos de Relacionamentos, Papéis e Restrições Estruturais 47

momento — um empregado pode gerenciar somente um departamento, e um departamento pode ter apenas um gerente nessa empresa. O tipo relacionamento TRABALHA_EM (Figura 3.13) tem razão de cardinalidade M:N, porque a regra do minimundo é que um empregado pode trabalhar em diversos projetos, e um projeto pode ter diversos empregados.

As razões de cardinalidade para relacionamentos binários são representadas nos diagramas ER pela exibição de 1, M ou N nos losangos, como mostrado na Figura 3.2.

FIGURA 3.12 Um relacionamento GERENCIA 1:1.
TRABALHAREM

FIGURA 3.13 Um relacionamento TRABALHA_EM, M:N.

Restrições de Participação e Dependências de Existência. A restrição de participação determina se a existência de uma entidade depende de sua existência relacionada à outra entidade, pelo tipo relacionamento. Essa restrição determina o número *mínimo* de instâncias de relacionamento em que cada entidade pode participar, e também é chamada restrição de cardinalidade mínima. Há dois tipos de restrições de participação — total e parcial — que ilustramos como exemplo. Se uma empresa adota a política de que todo empregado deve trabalhar para um departamento, então uma entidade empregado pode existir apenas se participar de, pelo menos, uma instância de relacionamento TRABALHA_PARA (Figura 3.9). Assim, a participação

de EMPREGADO em TRABALHA_PARA é chamada **participação total**, significando que toda entidade no 'conjunto total' de entidades empregados deve estar relacionada a uma entidade departamento, via TRABALHA_PARA. A participação total também é conhecida como **dependência de existência**. Na Figura 3.12 não esperamos que todo empregado gerencie um departamento; dessa forma, a participação de EMPREGADO no tipo relacionamento GERENCIA é **parcial**, significando que *algumas* ou 'parte do conjunto das' entidades estão relacionadas a algumas entidades departamento, via GERENCIA, mas não necessariamente todas elas. Iremos nos referir à razão de cardinalidade e restrições de participação, juntas, como **restrições estruturais** de um tipo relacionamento.

Nos diagramas ER, a participação total (ou dependência de existência) é exibida como uma *linha dupla* conectada ao tipo entidade participante do relacionamento, enquanto a participação parcial é representada por uma *linha única* (Figura 3.2).

3.4.4 Atributos de Tipos Relacionamento

Os tipos relacionamento também podem ter atributos, similares àqueles dos tipos entidade. Por exemplo, para registrar o número de horas semanais que um empregado trabalha em um determinado projeto, podemos incluir, na Figura 3.13, um atributo Horas para o tipo relacionamento TRABALHA_EM. Outro exemplo seria incluir a data em que o gerente começou a gerenciar o departamento por meio de um atributo DataInicio para o tipo relacionamento GERENCIA da Figura 3.12.

Observamos que os atributos de tipos relacionamento 1:1 ou 1:N podem ser migrados para um dos tipos entidade participantes. Por exemplo, o atributo DataInicio para o relacionamento GERENCIA pode ser um atributo tanto de EMPREGADO como de DEPARTAMENTO, embora conceitualmente pertença à GERENCIA. Isso é porque GERENCIA é um relacionamento 1:1, então, toda entidade departamento ou empregado participa de, *pelo menos*, uma instância do relacionamento. Portanto, o valor do atributo DataInicio pode ser determinado pela entidade departamento ou pela entidade empregado (gerente).

Para um tipo relacionamento 1:N, um atributo do relacionamento pode ser migrado *apenas* para o tipo entidade do lado N do relacionamento. Por exemplo, na Figura 3.9, se o relacionamento TRABALHA_PARA tem também um atributo DataInicio, que indica quando um empregado começou a trabalhar para o departamento — esse atributo pode ser incluído como um atributo de EMPREGADO. Isso porque cada empregado trabalha apenas para um departamento, portanto, participa de pelo menos uma instância de relacionamento de TRABALHA_PARA. Em ambos os tipos relacionamento, 1:1 e 1:N, a decisão de onde o atributo de relacionamento deve ser colocado — quer seja como um atributo do tipo relacionamento, quer seja como um atributo de um tipo entidade participante — é determinada, subjetivamente, pelo projetista do esquema.

Para tipos relacionamento M:N, alguns atributos são determinados pela *combinação de entidades participantes* de uma instância de relacionamento, e não por uma entidade única. Esses atributos *devem ser especificados como atributos de relacionamento*. Um exemplo é o atributo Horas, no relacionamento M:N TRABALHA_EM (Figura 3.13); o número de horas que um empregado trabalha em um projeto é determinado por uma combinação empregado-projeto, e não separadamente por qualquer uma das duas entidades.

3.5 TIPO ENTIDADE FRACA

Tipos entidade que não têm seus próprios atributos-chave são chamados **tipos entidade fraca**. Em contraste, **tipos entidade regular**, que têm um atributo-chave — os quais englobam todos os exemplos discutidos até aqui —, são chamados **tipos entidade forte**. Entidades, que pertencem a um tipo entidade fraca, são identificadas por estarem relacionadas a entidades específicas do outro tipo entidade, por meio da combinação com valores de seus atributos. Chamamos esse outro **tipo entidade identificador** ou **tipo entidade proprietária**, e chamamos o tipo relacionamento entre o tipo entidade fraca e seu tipo proprietário de **relacionamento identificador** do tipo entidade fraca. Um tipo entidade fraca sempre possui uma *restrição de participação total* (dependência de existência) em relação a seu relacionamento identificador, porque uma entidade fraca não poderá ser identificada sem um tipo proprietário. Porém, nem toda a dependência de existência resulta em um tipo entidade fraca. Por exemplo, uma entidade CARTEIRA_HABILITACAO não poderá existir a menos que esteja relacionada a uma entidade PESSOA, embora tenha sua própria chave (NumeroLicenca) e conseqüentemente não seja uma entidade fraca.

Considere o tipo entidade DEPENDENTE, relacionado a EMPREGADO, que é usado para manter o controle dos dependentes de cada empregado por meio de um relacionamento 1:N (Figura 3.2). Os atributos de DEPENDENTE são Nome (o primeiro nome do dependente), DataNascimento, Sexo e Parentesco (com o empregado). Dois dependentes de *dois empregados distintos* poderão ter, por casualidade, os mesmos valores para Nome, DataNascimento, Sexo e Parentesco, mas ainda assim serão entidades distintas. Eles só serão identificados como entidades distintas depois de determinar a *entidade empregado em particular* à

11 O tipo entidade identificador também é chamado algumas vezes tipo entidade pai ou tipo entidade dominante.

12 O tipo entidade fraca também é chamado, algumas vezes, tipo entidade filho ou tipo entidade subordinada.

3.6 Refinando o Projeto ER para o Banco de Dados Empresa 49

qual cada dependente estiver relacionado. É dito que cada entidade empregado é **proprietária** das entidades dependentes que estão relacionadas a ela.

Um tipo entidade fraca normalmente tem uma **chave parcial**, que é um conjunto de atributos que identifica, de modo exclusivo, as entidades fracas que estão relacionadas a uma mesma entidade proprietária. Em nosso exemplo, se assumirmos que nenhum par de dependentes do mesmo empregado poderá ter o mesmo primeiro nome, o atributo Nome de DEPENDENTE será sua chave parcial. No pior caso, o atributo composto por todos os atributos da entidade fraca será a chave parcial.

Em diagramas ER, tanto o tipo entidade fraca quanto seu relacionamento identificador serão identificados pela borda dupla no retângulo e no losango (Figura 3.2). O atributo chave parcial será sublinhado por uma linha sólida ou pontilhada.

Algumas vezes, os tipos entidade fraca poderiam ser representados por atributos complexos (compostos, multivalorados). No exemplo precedente, poderíamos especificar um atributo Dependente multivalorado para EMPREGADO que seria um tipo atributo composto por Nome, DataNascimento, Sexo e Parentesco. A escolha de qual representação usar é feita pelo projetista do banco de dados. Um critério que pode ser usado é, se houver muitos atributos, escolher a representação de tipo entidade fraca. Se a entidade fraca participar independentemente de outros tipos relacionamento, além de seu tipo relacionamento identificador, então *não* deveria ser modelada como um atributo complexo.

Em geral, pode ser definido qualquer número de níveis de tipos entidade fraca; um tipo entidade proprietário pode ser um tipo entidade fraca. Além disso, um tipo entidade fraca pode ter mais que um tipo entidade identificador e um tipo relacionamento identificador de grau mais alto que dois, como ilustraremos na Seção 4.7.

3.6 REFINANDO O PROJETO ER PARA O BANCO DE DADOS EMPRESA

Poderemos, agora, refinar o projeto de banco de dados da Figura 3.8 mudando os atributos que representam relacionamentos para tipos relacionamento. A razão de cardinalidade e a restrição de participação de cada tipo relacionamento é determinada pelos requisitos listados na Seção 3.2. Se alguma razão de cardinalidade ou dependência não puder ser determinada por meio dos requisitos, os usuários deverão ser consultados para determinar essas restrições estruturais. Em nosso exemplo, especificamos os tipos relacionamento seguintes:

1. GERENCIA, tipo relacionamento 1:1 entre EMPREGADO e DEPARTAMENTO. A participação de EMPREGADO é parcial. A participação de DEPARTAMENTO não está clara nas exigências. Questionamos os usuários que disseram que um departamento tem de ter sempre um gerente, o que indica participação total. O atributo DataInicio é designado para esse tipo relacionamento.
2. TRABALHA_PARA, um tipo relacionamento 1:N entre DEPARTAMENTO e EMPREGADO. Ambas as participações são totais.
3. CONTROLE, um tipo relacionamento 1:N entre DEPARTAMENTO e PROJETO. A participação de PROJETO é total, a participação de DEPARTAMENTO foi determinada como parcial, depois de consulta aos usuários, indicando que alguns departamentos podem não controlar nenhum projeto.
4. SUPERVISÃO, um tipo relacionamento 1:N entre EMPREGADO (no papel de supervisor) e EMPREGADO (no papel de supervisionado). Ambas as participações foram determinadas como parciais, depois que os usuários indicaram que nem todo empregado é um supervisor e nem todo empregado tem um supervisor.
5. TRABALHA_EM, determinado como um tipo relacionamento M:N, com o atributo Horas, depois de os usuários indicarem que um projeto pode alocar vários empregados. Ambas as participações foram determinadas como totais.
6. DEPENDE_DE, um tipo relacionamento 1:N entre EMPREGADO e DEPENDENTE, que também é o relacionamento identificador para o tipo entidade fraca DEPENDENTE. A participação de EMPREGADO é parcial, enquanto a de DEPENDENTE é total.

Depois de especificar os seis tipos relacionamento anteriores, removeremos, dos respectivos tipos entidade da Figura 3.8, todos os atributos que foram refinados em relacionamentos. Englobam: o Gerente e DataInicioGerencia de DEPARTAMENTO; DepartamentoControlador de PROJETO; Departamento, Supervisor, e TrabalhaEm de EMPREGADO; e Empregado de DEPENDENTE. É importante reduzir ao máximo a redundância quando projetamos o esquema conceitual de um banco de dados. Se alguma redundância for desejada no nível de armazenamento ou no nível de visão de usuário, ela poderá ser introduzida depois, conforme discutido na Seção 1.6.1.

13 A chave parcial às vezes é chamada discriminador.

14 As regras do minimundo que determinam as restrições são, às vezes, chamadas *regras de negócio*, uma vez que são determinadas pelo 'negócio', ou organização, que utilizará o banco de dados.

3.7 DIAGRAMAS ER, CONVENÇÕES DE NOMENCLATURA E DECISÕES DE PROJETO

3.7.1 Resumo das Notações para Diagramas ER

As figuras 3.9 a 3.13 ilustram exemplos da participação de tipos entidade em tipos relacionamento exibindo suas extensões — instâncias, em particular, de entidades, e instâncias de relacionamentos nos conjuntos de entidades e nos conjuntos de relacionamentos. Em diagramas ER, a ênfase está na representação do esquema em vez das instâncias. Isso é útil em projetos de banco de dados porque um esquema de banco de dados raramente é alterado, enquanto os conteúdos dos conjuntos de entidades frequentemente se alteram. Além disso, o esquema é normalmente mais fácil de exibir que toda a extensão de um banco de dados, pois é muito menor.

A Figura 3.2 mostra o esquema ER do banco de dados EMPRESA como um diagrama ER. Revisaremos agora toda a notação do diagrama ER. Tipos entidade como EMPREGADO, DEPARTAMENTO e PROJETO são mostrados em retângulos. Tipos relacionamento como TRABALHA_PARA, GERENCIA, CONTROLA e TRABALHA_EM são mostrados em losangos unidos aos tipos entidade participantes por linhas cheias. Os atributos são mostrados em elipses, e cada atributo é fixado diretamente a seu tipo entidade ou tipo relacionamento. O atributo componente de um atributo composto é anexado à representação oval do atributo composto, como ilustrado pelo atributo Nome de EMPREGADO. São mostrados atributos multivalorados em elipses duplas, como ilustrado pelo atributo Localização de DEPARTAMENTO. Atributos-chave são sublinhados. Atributos derivados são mostrados em elipses pontilhadas, como ilustrado pelo *NumerodeEmpregados*, atributo de DEPARTAMENTO.

Tipos entidade fraca são diferenciados por meio de retângulos com bordas duplas e por terem seu relacionamento identificador colocado em losangos com bordas duplas, como ilustrado pelo tipo entidade DEPENDENTE e pelo tipo relacionamento identificador DEPENDE_DE. A chave parcial do tipo entidade fraca é sublinhada com uma linha pontilhada.

Na Figura 3.2 a razão de cardinalidade de cada tipo relacionamento binário é especificada anexando um 1, M ou N a cada linha de participação. A razão de cardinalidade entre DEPARTAMENTO:EMPREGADO em GERENCIA é de 1:1, enquanto é 1:N para DEPARTAMENTO:EMPREGADO em TRABALHA_PARA, e M:N para TRABALHA_EM. A restrição de participação é especificada por uma única linha para participação parcial e por meio de linhas duplas para participação total (dependência de existência).

Na Figura 3.2 mostramos os nomes dos papéis para o tipo relacionamento SUPERVISÃO, pois o tipo entidade EMPREGADO faz ambos os papéis naquele relacionamento. Note que a cardinalidade é 1:N de supervisor para supervisionado, porque cada empregado no papel de supervisionado tem, no máximo, um supervisor direto, enquanto um empregado, no papel de supervisor, poderá supervisionar zero ou mais empregados.

A Figura 3.14 resume as convenções para diagramas ER.

3.7.2 Denominação dos Construtores dos Esquemas

Ao projetar um esquema de banco de dados, a escolha dos nomes para tipos entidade, atributos, relacionamentos, tipos e (particularmente) papéis nem sempre é direta. Deveriam ser escolhidos nomes que carregassem, tanto quanto possível, os significados dos diferentes construtores do esquema. Optamos por usar *nomes no singular* para os tipos entidade em vez do plural, porque o nome do tipo entidade se aplica a cada entidade em particular pertencente àquele tipo. Em nossos diagramas ER usaremos como convenção letras maiúsculas para tipos entidade e tipos relacionamento, a primeira letra maiúscula para os nomes dos atributos e letras minúsculas para os nomes dos papéis. Usamos essa convenção na Figura 3.2.

Como prática geral, dada uma descrição narrativa dos requisitos do banco de dados, os *substantivos* do texto tendem a originar nomes de tipos entidade e os *verbos* tendem a indicar nomes de tipos relacionamento. Os nomes dos atributos geralmente surgem de substantivos adicionais que descrevem os substantivos correspondentes aos tipos entidade.

Outra observação sobre a nomenclatura envolve a escolha dos nomes dos relacionamentos binários do esquema ER de modo a torná-los legíveis para a leitura da direita para a esquerda e de cima para baixo. Em geral, seguimos essa recomendação na Figura 3.2. Temos uma exceção nessa convenção na Figura 3.2 — o tipo relacionamento DEPENDENTE_DE é lido de baixo para cima. Quando descrevemos esse relacionamento podemos dizer que as entidades DEPENDENTE (tipo entidade de baixo) são DEPENDENTES_DE (nome do relacionamento) um EMPREGADO (tipo entidade de cima). Para mudar a leitura de cima para baixo, poderíamos denominar o tipo relacionamento como TEM_DEPENDENTE, que se leria como segue: uma entidade EMPREGADO (tipo entidade de cima) TEM DEPENDENTE (nome do relacionamento) de tipo DEPENDENTE (tipo entidade de baixo). Note que esse problema surge porque cada relacionamento binário pode ser descrito a partir de qualquer um dos dois tipos entidade participantes, como discutido no início da Seção 3.4.

3.7.3 Decisões de Projetos para o Projeto Conceitual ER

Às vezes, é difícil decidir se um determinado conceito no minimundo deveria ser modelado como um tipo entidade, um atributo ou um tipo relacionamento. Nesta seção, daremos um breve resumo das diretrizes sobre qual construtor deveria ser escolhido em situações particulares.


<u>Símbolo</u>	<u>Significado</u>
	ENTIDADE
	FRACA
	RELACIONAMENTO
	IDENTIFICADOR DE RELACIONAMENTO
	ATRIBUTO-CHAVE
	ATRIBUTO MULTIVALORADO
	ATRIBUTO COMPOSTO
	ATRIBUTO DERIVADO
	PARTICIPAÇÃO TOTAL DE E, EM R
N	RAZÃO DE CARDINALIDADE 1:/V PARA E_1E_2 EM R
	RESTRIÇÃO ESTRUTURAL (MIN.MAX) DA PARTICIPAÇÃO DE E EM R

FIGURA 3.14 Resumo da notação para diagramas ER.

Em geral, o processo de projeto do esquema deveria ser considerado um processo iterativo de refinamento, no qual um projeto inicial é criado e então iterativamente refinado até que o projeto mais satisfatório seja alcançado. Alguns dos refinamentos mais frequentemente usados são:

- Um conceito pode ser modelado primeiro como um atributo, e então pode ser refinado em um relacionamento porque foi determinado que o atributo é uma referência a outro tipo entidade. É comum que um par de tais atributos, que são o inverso um do outro, sejam refinados em um relacionamento binário. Discutiremos esse tipo de refinamento em detalhes na Seção 3.6.
- Analogamente, um atributo existente em vários tipos entidade pode ser promovido ou elevado a um tipo entidade independente. Por exemplo, suponha que vários tipos entidade de um banco de dados UNIVERSIDADE, como ALUNO, INSTRUTOR e CURSO, tenham, cada um, um atributo Departamento no projeto inicial; o projetista poderia criar, então, um tipo entidade DEPARTAMENTO com um único atributo NomeDepto e relacioná-lo aos três tipos entidade (ALUNO, INSTRUTOR e CURSO) por meio de relacionamentos apropriados. Outros atributos/relacionamentos de DEPARTAMENTO poderão aparecer depois.
- Um refinamento inverso para o caso anterior poderá ser aplicado — por exemplo, se um tipo entidade DEPARTAMENTO existir no projeto inicial com um único atributo NomeDepto, e estiver relacionado a somente um tipo entidade, ALUNO. Nesse caso, DEPARTAMENTO pode ser reduzido ou rebaixado a um atributo de ALUNO.
- No Capítulo 4, discutiremos outros refinamentos relativos à especialização/generalização e a relacionamentos de graus mais altos. O Capítulo 12 discute refinamentos adicionais *top-down* (cima-para-baixo) e *bottom-up* (baixo-para-cima), que são comuns em grandes projetos de esquemas conceituais.

3.7.4 Notações Alternativas para Diagramas ER

Há muitas notações diagramáticas alternativas para a exibição de diagramas ER. O Apêndice A dá algumas das notações mais usadas. Na Seção 3.8 introduziremos a notação da Universal Modeling Language (UML — Linguagem de Modelagem Universal) para os diagramas de classe, os quais foram propostos como padrão para a modelagem conceitual de objetos.

Nesta seção, descreveremos uma notação ER alternativa para especificar restrições estruturais em relacionamentos. Essa notação envolve a associação de um par de números inteiros (min, max) a cada *participação* de um tipo entidade E em um tipo relacionamento R, onde $0 < \text{min} < \text{max}$ e $\text{max} > 1$. Os números significam que para cada entidade e em E, e deve participar em pelo menos min e no máximo em max relacionamentos em R a *qualquer tempo*. Nesse método, min = 0 indica participação parcial, enquanto min > 0 indica participação total.

A Figura 3.15 mostra o esquema do banco de dados EMPRESA que usa a notação (min, max). Normalmente, usa-se a notação cardinalidade/linha-simples/linha-dupla ou a notação (min, max). A notação (min, max) é mais precisa e podemos usá-la facilmente para especificar restrições estruturais para tipos relacionamento de *qualquer grau*. Porém, isso não é suficiente para a especificação de algumas restrições-chave em relacionamentos de alto grau, como será discutido na Seção 4.7.

A Figura 3.15 também exhibe todos os nomes dos papéis para o esquema de banco de dados EMPRESA.

3.8 NOTAÇÃO PARA DIAGRAMAS DE CLASSE UML

A metodologia UML está sendo extensivamente usada em projetos de software e possui vários tipos de diagramas utilizados nesses projetos. Apresentamos brevemente, aqui, apenas os diagramas de classe da UML, e os comparamos aos diagramas ER. Sob certos aspectos, os diagramas de classe podem ser considerados uma notação alternativa aos diagramas ER. A notação UML e os conceitos adicionais são apresentados na Seção 4.6 e no Capítulo 12. A Figura 3.16 mostra como o esquema do banco de dados ER EMPRESA, da Figura 3.15, pode ser exibido usando notação UML de diagrama de classe. Os *tipos entidade* da Figura 3.15 são modelados como *classes* na Figura 3.16. Uma *entidade* no ER corresponde a um *objeto* na UML.

Em diagramas de classe UML, uma classe é apresentada como uma caixa (Figura 3.16) com três seções: a seção superior fornece o nome da classe, a seção do meio os atributos para os objetos individuais da classe, e a última seção mostra as operações que podem ser aplicadas a esses objetos. Operações *não* são especificadas em diagramas ER. Considere a classe EMPREGADO na Figura 3.16. Seus atributos são Nome, SSN, DataN, Sexo, Endereço e Salário. Opcionalmente, se desejado, o projetista pode especificar o domínio de um atributo colocando dois pontos (:) seguidos do nome do domínio ou descrição, conforme ilustrado pelos atributos Nome, Sexo e DataN de EMPREGADO na Figura 3.16. Um atributo composto é modelado como um domínio estruturado, segundo ilustrado pelo atributo Nome de EMPREGADO. Um atributo multivalorado será modelado, geralmente, como uma classe separada, conforme ilustrado pela classe LOCALIZAÇÃO, na Figura 3.16.

15 Em algumas notações em particular, como as usadas em metodologias de modelagem de objetos, como a UML, o (min, max) é colocado em *lados opostos* aos que mostramos. Por exemplo, para o relacionamento TRABALHA_PARA, da Figura 3.15, o (1,1) estaria no lado de DEPARTAMENTO, e o (4,N) no lado de EMPREGADO. Aqui usamos a notação original de Abrial (1974).

dependente (1,1)

DEPENDENTE

Figura 3.15 Diagrama ER para o esquema EMPRESA. As restrições estruturais são especificadas usando a notação (min,max).

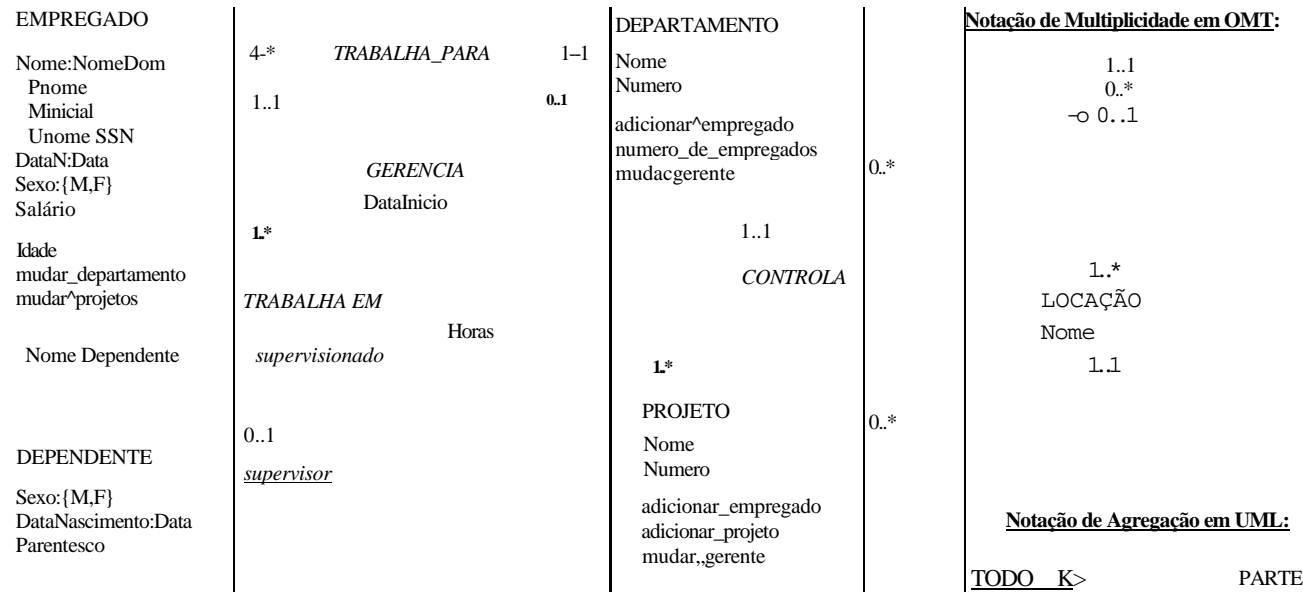


FIGURA 3.16 O esquema conceitual EMPRESA em notação de diagrama de classe UML.

Os tipos relacionamento são chamados associações na terminologia UML e as instâncias de relacionamento são conhecidas como links. Uma associação binária (tipo relacionamento binário) é representada por uma linha conectada às classes participantes (tipos entidade) e pode, opcionalmente, ter um nome. Um atributo de relacionamento, chamado atributo de link, é colocado em uma caixa, que está conectada à linha de associação por uma linha tracejada. A notação (min, max), descrita na Seção 3.7.4, é usada para especificar as restrições de relacionamento, que são chamadas multiplicidades na terminologia UML. As multiplicidades são especificadas na forma *min..max* e um asterisco (*) indica que não há limite máximo na participação. No entanto, as multiplicidades são colocadas *nas extremidades opostas do relacionamento*, quando comparadas com a notação discutida na Seção 3.7.4 (confrontar com as figuras 3.16 e 3.15). Em UML, um asterisco único indica uma multiplicidade de 0..*, e um 1 único indica uma multiplicidade de 1..1. Um relacionamento recursivo (Seção 3.4.2) é chamado, em UML, de uma associação reflexiva, e os nomes de papéis — bem como as multiplicidades — são colocados nas extremidades opostas de uma associação, quando comparados com a colocação dos nomes de papéis na Figura 3.15.

Em UML há dois tipos de relacionamentos: a associação e a agregação. A agregação é um meio para representar um relacionamento entre um objeto e suas partes componentes, e possui uma notação diagramática diferente. Na Figura 3.16 modelamos as localizações de um departamento e a localização única de um projeto como agregações. Entretanto, a agregação e a associação não têm propriedades estruturais distintas, e a escolha sobre qual tipo de relacionamento usar é um pouco subjetiva. No modelo ER, ambas são representadas como relacionamentos.

A UML também faz distinção entre as associações (ou as agregações) unidirecional e bidirecional. No caso unidirecional, a linha que conecta as classes é exibida com uma seta para indicar que é necessária apenas uma direção para acessar os objetos relacionados. Se nenhuma seta for exibida, o caso bidirecional será assumido, pois é o padrão. Por exemplo, se sempre esperamos acessar o gerente de um departamento começando pelo objeto DEPARTAMENTO, desenharíamos a linha de associação representando a associação GERENCIA com uma seta de DEPARTAMENTO para EMPREGADO. Além disso, as instâncias de relacionamento podem ser especificadas para serem ordenadas. Por exemplo, podemos especificar que os objetos de empregado relacionados a cada departamento pela associação (relacionamento) TRABALHA_PARA deveriam estar ordenados pelos valores de seu atributo DataN. Os nomes de associação (relacionamento) são *opcionais* na UML, e os atributos de relacionamento são exibidos em uma caixa conectada, por uma linha tracejada, à linha representando a associação/agregação (DataInicio e Horas na Figura 3.16).

As operações em cada classe derivam dos requisitos funcionais da aplicação, conforme discutimos na Seção 3.1. Geralmente é suficiente especificar, de início, os nomes de operação para as operações lógicas que se espera serem aplicadas aos objetos individuais de uma classe, conforme mostrado na Figura 3.16. A medida que o projeto é refinado, mais detalhes são adicionados, como os tipos de argumentos exatos (parâmetros) para cada operação, além da descrição funcional de cada operação. A UML tem *descrições funcionais* e *diagramas de sequência* para especificar alguns detalhes de operação, mas estes estão além do escopo da nossa discussão. O Capítulo 12 apresentará alguns desses diagramas.

As entidades fracas podem ser modeladas usando-se a construção chamada associação qualificada (ou agregação qualificada) na UML. Isso permite representar os relacionamentos identificadores e a chave parcial, que é colocada em uma caixa conectada à classe forte. Isso está ilustrado na Figura 3.16 pela classe DEPENDENTE e sua agregação qualificada com EMPREGADO. A chave parcial NomeDependente é chamada discriminador na terminologia UML, uma vez que seu valor distingue os objetos associados ao (relacionados ao) mesmo EMPREGADO. As associações qualificadas não estão restritas à modelagem de entidades fraca e podem ser usadas para modelar outras situações em UML.

3.9 RESUMO

Neste capítulo apresentamos os conceitos de modelagem presentes em um modelo de dados conceitual de alto nível, o modelo Entidade-Relacionamento (ER). Iniciamos com a discussão do papel que um modelo de dados de alto nível tem no processo de projeto do banco de dados então apresentamos exemplos dos requisitos para o banco de dados EMPRESA, que é um dos exemplos usados no decorrer deste livro. Definimos, então, os conceitos básicos de entidades e atributos do modelo ER. Discutimos os valores *nulls* (*nulos*) e apresentamos os vários tipos de atributos que podem ser aninhados para produzir os atributos complexos:

- Simples ou atômico
- Composto
- Multivalorado

3.9 Resumo 55

Discutimos também brevemente os atributos armazenados *versus* derivados. Então discutimos os conceitos do modelo ER para o esquema ou 'conotação':

- Tipos entidade e seus conjuntos de entidade correspondentes
- Atributos-chave de tipos entidade
- Conjunto de valores (domínios) de atributos
- Tipos relacionamento e seus conjuntos de relacionamento correspondentes
- Papéis desempenhados pelos tipos entidade nos tipos relacionamento

Apresentamos dois métodos para a especificação de restrições estruturais em tipos relacionamento. O primeiro método distingue dois tipos de restrições estruturais:

- Razões de cardinalidade (1:1, 1:N, M:N para os relacionamentos binários)
- Restrições de participação (total, parcial)

Observamos que uma alternativa, para representar as restrições estruturais, é especificar os números mínimo e máximo (min, max) na participação de cada tipo entidade em um tipo relacionamento. Discutimos tipos entidade fraca e os conceitos relacionados de tipos entidade forte,* identificando tipos relacionamento e atributos-chave parciais.

Os esquemas Entidade-Relacionamento podem ser representados na forma de diagramas ER. Mostramos como projetar um esquema ER para o banco de dados EMPRESA definindo, primeiro, os tipos entidade e seus atributos então refinando o projeto para incluir os tipos relacionamento. Exibimos em seguida o diagrama ER para o esquema do banco de dados EMPRESA. Finalmente, alguns dos conceitos básicos de diagramas de classe UML e como se relacionam com os conceitos do modelo ER.

Os conceitos de modelagem ER que apresentamos até aqui — tipos entidade, tipos relacionamento, atributos-chave e restrições estruturais — podem modelar as aplicações tradicionais de banco de dados para o processamento de dados de empresas. Entretanto, atualmente, as aplicações mais complexas — como projetos de engenharia, sistemas de informações médicas ou telecomunicações — exigem os conceitos adicionais, se desejamos modelá-los com maior exatidão. Discutiremos esses conceitos avançados de modelagem no Capítulo 4. Também nesse capítulo descreveremos os tipos relacionamento ternário e de grau mais alto com mais detalhes, e discutiremos as circunstâncias anteriores, quando são diferenciadas dos relacionamentos binários.

Questões de Revisão

- 3.1 Discuta o papel do modelo de dados de alto nível no processo de projeto de um banco de dados.
- 3.2 Liste os vários casos nos quais o uso de um valor *null* (nulo) seria apropriado.
- 3.3 Defina os seguintes termos: *entidade*, *atributo*, *valor de atributo*, *instância de relacionamento*, *atributo composto*, *atributo multivalorado*, *atributo derivado*, *atributo complexo*, *atributo-chave*, *conjunto de valores (domínio)*.
- 3.4 O que é um tipo entidade? O que é um conjunto de entidades? Explique as diferenças entre uma entidade, um tipo entidade e um conjunto de entidades.
- 3.5 Explique a diferença entre um atributo e um conjunto de valores.
- 3.6 O que é um tipo relacionamento? Explique as diferenças entre uma instância de relacionamento, um tipo relacionamento e um conjunto de relacionamentos.
- 3.7 O que é um papel de participação? Quando é necessário usar os nomes de papéis na descrição de tipos relacionamento?
- 3.8 Descreva as duas alternativas para a especificação de restrições estruturais em tipos relacionamento. Quais são as vantagens e desvantagens de cada uma?
- 3.9 Sob quais condições um atributo de um tipo relacionamento binário pode ser migrado para se tornar um atributo de um dos tipos entidade participantes?
- 3.10 Quando pensamos em relacionamentos como atributos, quais são os conjuntos de valores desses atributos? Qual classe dos modelos de dados está baseada nesse conceito?
- 3.11 O que se entende por um tipo relacionamento recursivo? Dê alguns exemplos de tipos relacionamento recursivos.
- 3.12 Quando o conceito de entidade fraca é usado na modelagem de dados? Defina os termos *tipo entidade forte*, *tipo entidade fraca*, *tipo relacionamento identificador* e *chave parcial*.

O termo *owner* empregado pelo autor seria normalmente traduzido como *proprietário*, entretanto, como em outras ocorrências, optamos pelo termo mais usual no Brasil: *forte*. (N. de T.)

- 3.13 Um relacionamento identificador de um tipo entidade fraca pode ser de grau maior que dois? Dê exemplos para ilustrar sua resposta.
- 3.14 Discuta as convenções para a exibição de um esquema ER como um diagrama ER.
- 3.15 Discuta as convenções utilizadas para os nomes nos diagramas de esquema ER.

Exercícios

- 3.16 Considere o seguinte conjunto de requisitos para um banco de dados universitário usado para gerar o histórico escolar dos alunos (relatório oficial fornecido pela escola). Este é similar, mas não idêntico, ao banco de dados mostrado na Figura 1.2:
- A universidade mantém, para cada aluno, nome, número do aluno, número do seguro social, endereço e telefone atuais, endereço e telefone permanentes, data de nascimento, sexo, turma (calouro, segundalista, ..., graduado), departamento principal, departamento secundário (se houver) e programa de graduação (B.A., B.S., ..., Ph.D.). Algumas aplicações de usuários precisam de dados como cidade, estado e CEP do endereço permanente do estudante, além do último nome dele. O número do seguro social e o número do aluno têm valores únicos para cada aluno.
 - Cada departamento é descrito por um nome, código do departamento, número do escritório, telefone do escritório e faculdade. O nome e código têm valores únicos para cada departamento.
 - Cada curso tem um nome do curso, descrição, número do curso, número de horas semestrais, nível e departamento responsável. O valor do número do curso é único para cada curso.
 - Cada disciplina tem um instrutor, semestre, ano, curso e número da disciplina. O número da disciplina distingue as disciplinas do mesmo curso que são lecionadas durante o mesmo semestre/ano; seus valores são 1, 2, 3,..., até o número total de disciplinas lecionadas durante cada semestre.
 - Um relatório da graduação contém um aluno, disciplina, letra da graduação e número (0, 1, 2, 3 ou 4).
- Projete um esquema ER para essa aplicação e desenhe um diagrama ER para esse esquema. Especifique os atributos-cha-ve para cada tipo entidade e as restrições estruturais em cada tipo relacionamento. Observe qualquer requisito não especificado e faça as suposições apropriadas para tornar a especificação completa.
- 3.17 Os atributos compostos e multivalorados podem ser aninhados em qualquer número de níveis. Suponha que queiramos projetar um atributo para o tipo entidade ALUNO para manter o controle de sua educação prévia em outros cursos. Esse atributo terá uma entrada para cada faculdade freqüentada anteriormente, e cada uma dessas entradas será composta pelo nome da faculdade, datas de início e término, entradas dos títulos (graus concedidos pela faculdade, se houver), bem como de histórico escolar (cursos completados na faculdade, se houver). Cada entrada de titulação (grau) contém o nome, o mês e o ano em que foi concedida e cada entrada de histórico inclui nome de curso, semestre, ano e título obtido. Projete um atributo para controlar essa informação. Use as convenções da Figura 3.5.
- 3.18 Mostre um projeto alternativo para o atributo descrito no Exercício 3.17 que use apenas os tipos entidade (incluindo os tipos entidade fraca, se necessário) e os tipos relacionamento.
- 3.19 Considere o diagrama ER da Figura 3.17 que mostra um esquema simplificado para as reservas aéreas. Extraia do diagrama ER os requisitos e restrições que produziram esse esquema. Tente ser tão preciso quanto possível na especificação de seus requisitos e restrições.
- 3.20 Nos capítulos 1 e 2 discutimos o ambiente e os usuários de um banco de dados. Podemos considerar muitos tipos entidade para descrever um ambiente, como SGBD, banco de dados armazenado, DBA e catálogo/dicionário de dados. Tente especificar todos os tipos entidade que podem descrever totalmente um sistema de banco de dados e seu ambiente então especifique os tipos relacionamento entre eles, e desenhe um diagrama ER para descrever o ambiente geral desse banco de dados.
- 3.21 Projete um esquema ER para o controle de informação sobre os votos coletados na Casa dos Representantes dos Estados Unidos durante as sessões congressionais dos dois anos correntes. O banco de dados necessita controlar o nome de cada Nome ESTADO dos Estados Unidos (por exemplo, Texas, Nova York, Califórnia) e incluir a Região do estado (cujo domínio é {Nordeste, Centro-Oeste, Sudeste, Sudoeste e Oeste}). Cada CONGRESSISTA da Casa de Representantes é descrito pelo Nome dele ou dela, mais o Distrito representado, a DataInício de quando o congressista foi eleito pela primeira vez, e o Partido político ao qual ele ou ela pertence (cujo domínio é {Republicano, Democrata, Independente, Outro}). O banco de dados mantém o controle de cada PROJETOLEI (isto é, lei proposta), incluindo o NomeProjetoLei, a Datado Voto no projeto de lei, se o projeto de lei é AprovadoReprovado (cujo domínio é {Sim, Não}) e o Responsável (o(a) congressista, que o propôs). O banco de dados mantém o controle de como cada congressista votou em cada projeto de lei (o do-

mínio do atributo voto é {Sim, Não, Absteve-se, Ausente}). Desenhe um diagrama do esquema ER para essa aplicação. Declare, claramente, qualquer suposição que você faça.

NOTA

- (1) UM TRECHO (SEGMENTO) É UM TRECHO DE VÔO SEM ESCALA
- (2) UMA INSTÂNCIA DE UM TRECHO É UMA OCORRÊNCIA PARTICULAR DE UM TRECHO EM UMA DATA ESPECÍFICA

POLTRONA

FIGURA 3.17 Um diagrama ER para um esquema de banco de dados COMPANHIA AÉREA.

- 3.22 Um banco de dados está sendo construído para manter o controle dos times e jogos de uma liga esportiva. Um time tem um número de jogadores, dos quais não são todos que participam de um determinado jogo. Deseja-se controlar os jogadores de cada time, que participam de cada jogo, as posições em que jogam e o resultado do jogo. Projete um diagrama de esquema ER para essa aplicação, declarando quaisquer suposições que você fizer. Escolha seu esporte favorito (por exemplo, futebol, beisebol, futebol americano).
- 3.23 Considere o diagrama ER mostrado na Figura 3.18 como parte de um banco de dados BANCO. Cada banco pode ter múltiplas agências e cada agência pode conter múltiplas contas e empréstimos.
- a. Liste os tipos entidade (não fracas) do diagrama ER.
 - b. Há um tipo entidade fraca? Se sim, dê o seu nome, chave parcial e o relacionamento identificador.
 - c. Quais restrições a chave parcial e o relacionamento identificador do tipo entidade fraca especificam nesse diagrama ?
 - d. Liste os nomes de todos os tipos relacionamento e especifique a restrição (min, max) em cada participação de um tipo entidade, em um tipo relacionamento. Justifique suas escolhas.

- e. Liste, concisamente, os requisitos do usuário que levaram a esse projeto de esquema ER.
- f. Suponha que cada cliente deva ter, pelo menos, uma conta, mas está limitado ao máximo de dois empréstimos de cada vez, e que uma agência bancária não pode ter mais de mil empréstimos. Como fazer as restrições (min, max) para essa situação?



FIGURA 3.18 Um diagrama ER para um esquema de banco de dados BANCO.

- 3.24 Considere o diagrama ER da Figura 3.19. Suponha que um empregado possa trabalhar em mais de dois departamentos ou possa não estar designado a qualquer departamento. Pressuponha que cada departamento deva ter um e possa ter mais de três números de telefone. Forneça as restrições (min, max) desse diagrama. *Declare, claramente, qualquer suposição adicional que você fizer.* Nesse exemplo, sob quais condições o relacionamento TEM_TELEFONE seria redundante?



TELEFONE

FIGURA 3.19 Parte de um diagrama ER para um banco de dados EMPRESA.

- 3.25 Considere o diagrama ER da Figura 3.20. Suponha que um curso possa ou não usar um livro didático, mas que um texto, por definição, seja um livro usado em um curso. Um curso não pode usar mais de cinco livros. Os instrutores ensinam em dois a quatro cursos. Forneça as restrições (min, max) desse diagrama. *Declare, claramente, qualquer suposição adicional que você fizer.* Se adicionarmos o relacionamento ADOTA entre INSTRUTOR e TEXTO, quais restrições (min, max) você colocaria nele? Por quê?
- 3.26 Leve em consideração um tipo entidade DISCIPLINA em um banco de dados UNIVERSIDADE, que descreve as disciplinas oferecidas nos cursos. Os atributos de DISCIPLINA são NumeroDisciplina, Semestre, Ano, NumeroCurso, Instrutor, NumSala (onde a disciplina é ensinada), Prédio (onde a disciplina é ensinada), DiasUteis (o domínio é a combinação possível de dias úteis nos quais uma disciplina pode ser oferecida {SEGQUASEX, SEGQUA, TERQUI etc.}) e Horas (o domínio é todo o período possível durante o qual as disciplinas são oferecidas {9-9:50 A.M., 10-10:50 A.M.,..., 3:30-4:50 P.M., 5:30-6:20 P.M. etc.}). Presuma que NumeroDisciplina seja único para cada curso, em uma combinação semestre/ano em particu-

lar (isto é, se um curso é oferecido várias vezes em um semestre em particular, as disciplinas oferecidas são numeradas 1, 2, 3 etc). Há diversas chaves compostas para DISCIPLINA, e alguns atributos são componentes de mais de uma chave. Identifique três chaves compostas e mostre como elas podem ser representadas em um diagrama de esquema ER.

INSTRUTOR

|

CURSO

TEXTO

FIGURA 3.20 Parte de um diagrama ER para um banco de dados CURSOS.

Bibliografia Seleccionada

O modelo Entidade-Relacionamento foi introduzido por Chen (1976) e em trabalhos relacionados em Schmidt e Swenson (1975), Wiederhold e Elmasri (1979), e Senko (1975). Desde essa época, numerosas modificações têm sido sugeridas ao modelo ER. Temos incorporado algumas delas em nossas apresentações. As restrições estruturais em relacionamentos são discutidas em Abrial (1974), Elmasri e Wiederhold (1980), e Lenzerini e Santucci (1983). Os atributos compostos e multivalorados são incorporados ao modelo ER em Elmasri *et al.* (1985). Embora não tenhamos discutido as linguagens para o modelo entida-de-relacionamento e suas extensões, tem havido diversas propostas para linguagens. Elmasri e Wiederhold (1981) propuseram a linguagem de consulta GORDAS para o modelo ER. Outra linguagem de consulta ER foi proposta por Markowitz e Raz (1983). Senko (1980) apresenta uma linguagem de consulta para o modelo DIAM, de Senko. Um conjunto formal de operações chamado álgebra ER foi exposto por Parent e Spaccapietra (1985). GogoUa e Hohenstein (1991) proporcionaram outra linguagem formal para o modelo ER. Campbell *et al.* (1985) forneceram um conjunto de operações ER e mostraram que elas são relacionalmente completas. Uma conferência para a disseminação de resultados de pesquisas relacionadas ao modelo ER tem acontecido desde 1979. A conferência, agora conhecida como a Conferência Internacional sobre a Modelagem Conceitual, aconteceu em Los Angeles (ER 1979, ER 1983, ER 1997), Washington, D.C. (ER 1981), Chicago (ER 1985), Dijon, França (ER 1986), Nova York (ER 1987), Roma (ER 1988), Toronto (ER 1989), Lausanne, Suíça (ER 1990), San Mateo, Califórnia (ER 1991), Karlsruhe, Alemanha (ER 1992), Arlington, Texas (ER 1993), Manchester, Inglaterra (ER 1994), Brisbane, Austrália (ER 1995), Cottbus, Alemanha (ER 1996), Cingapura (ER 1998), Salt Lake City, Utah (ER 1999), Yokohama, Japão (ER 2001) e Tampere, Finlândia (ER 2002). A última conferência foi realizada em Chicago em outubro de 2003.

Modelagem com Entidade-Relacionamento Estendido e UML

OS conceitos de modelagem ER discutidos no Capítulo 3 são suficientes para a representação de muitos esquemas para as aplicações de bancos de dados "tradicionais", que incluem, principalmente, as aplicações de processamento de dados no comércio e na indústria. Desde o final dos anos 70, entretanto, os projetistas de bancos de dados têm tentado projetar esquemas de bancos de dados mais exatos, que reflitam as propriedades e as restrições dos dados mais precisamente. Isso foi particularmente importante nas recentes aplicações da tecnologia de banco de dados, como os bancos de dados para o projeto de engenharia e manufatura (CAD/CAM), telecomunicações, sistemas de software complexos e Sistemas de Informação Geográfica — Geographic Information Systems (GIS), entre muitas outras. Esses tipos de bancos de dados têm requisitos mais complexos que a maioria das aplicações tradicionais. Isso levou ao desenvolvimento de conceitos adicionais de *modelagem semântica de dados*, que foram incorporados aos modelos de dados conceituais, como o modelo ER. Vários modelos semânticos de dados têm sido propostos na literatura. Muitos desses conceitos também foram desenvolvidos independentemente, em áreas relacionadas da ciência da computação, como nas áreas de **representação do conhecimento** da inteligência artificial e de **modelagem de objeto** da engenharia de software.

Neste capítulo, descreveremos os aspectos que têm sido propostos para modelos semânticos de dados e mostraremos como o modelo ER pode ser expandido para incluir esses conceitos, levando ao modelo **ER estendido** ou **EER**. Começaremos na Seção 4.1 pela incorporação de conceitos de *relacionamentos de classes/subclasses* e tipo de *herança* no modelo ER. Então, na Seção 4.2, adicionaremos os conceitos de *especialização* e *generalização*. Na Seção 4.3 discutiremos os vários tipos de *restrições* à especialização/generalização, e na Seção 4.4 mostraremos como a construção UNIÃO pode ser modelada pela inclusão do conceito de *categoria* no modelo EER. A Seção 4.5 apresenta um exemplo de um esquema do banco de dados UNIVERSIDADE no modelo EER, e resume os conceitos do modelo EER, fornecendo definições formais.

Na Seção 4.6 apresentaremos, então, a notação do diagrama de classes UML e os conceitos para a representação de especialização e generalização, e os comparamos, brevemente, aos conceitos e notação EER. Esta será uma continuação da Seção 3.8, que apresentou uma notação básica do diagrama de classe UML.

A Seção 4.7 discute algumas das mais complexas questões envolvidas na modelagem de relacionamentos ternários e graus superiores. Na Seção 4.8 discutiremos as abstrações fundamentais que são usadas como base para muitos modelos semânticos de dados. A Seção 4.9 resume o capítulo.

- 1 CAD/CAM significa projeto auxiliado por computador (CAD — *computer-aided design*)/manufatura auxiliada por computador (CAM — *computer-aided manufacturing*).
- 2 EER também tem sido chamado ER Expandido.
- * Cabe comentar para essa nota que, em inglês, há duas grafias para EER: Enhanced ER (ER melhorado, incrementado) e Extended ER (ER estendido, aumentado); usaremos ER Estendido. (N. de T.)

Para uma introdução detalhada de modelagem conceitual, o Capítulo 4 deve ser considerado uma continuação do Capítulo 3. Entretanto, se for desejada apenas uma introdução básica à modelagem ER, este capítulo pode ser omitido. Alternativamente, o leitor pode optar pular algumas, ou todas, as seções finais deste capítulo (seções 4.4 até 4.8).

4.1 SUBCLASSES, SUPERCLASSES E HERANÇA

O modelo EER (ER Estendido) engloba todos os conceitos de modelagem do modelo ER que foram apresentados no Capítulo 3. Além disso, inclui os conceitos de subclasse e **superclasse** e os conceitos relacionados de especialização e **generalização** (seções 4.2 e 4.3). Outro conceito incluído no modelo EER é o de uma **categoria** ou **tipo de união** (Seção 4.4), que é usado para representar uma coleção de objetos correspondente à *união* de objetos de diferentes tipos entidade. Associado a esses conceitos está o importante mecanismo de **herança de atributo e relacionamento**. Infelizmente, não existe terminologia-padrão para esses conceitos, então usaremos a mais comum. A terminologia alternativa é dada nas notas de rodapé. Descreveremos também uma técnica diagramática para a exibição desses conceitos quando aparecerem em um esquema EER. Chamamos os diagramas do esquema resultante ER **estendido** ou **diagramas EER**.

O primeiro conceito do modelo EER que estudaremos é o de uma **subclasse** de um tipo entidade. Conforme discutimos no Capítulo 3, um tipo entidade é usado para representar tanto um tipo *entidade* como o *conjunto de entidades* ou *coleção de entidades daquele tipo* existente no banco de dados. Por exemplo, o tipo entidade EMPREGADO descreve o tipo (isto é, os atributos e os relacionamentos) de cada entidade empregado e também se refere ao conjunto corrente das entidades EMPREGADO do banco de dados EMPRESA. Em muitos casos, um tipo entidade tem numerosos subgrupos dessas entidades, que são significativos e que necessitam ser representados explicitamente, em virtude de sua importância para a aplicação do banco de dados. Por exemplo, as entidades que são membros do tipo entidade EMPREGADO podem ser posteriormente agrupadas em SECRETÁRIA, ENGENHEIRO, GERENTE, TÉCNICO, EMPREGADO_ASSALARIADO, EMPREGADO_HORISTA, e assim por diante. O conjunto de entidades de cada um desses grupos é um subconjunto das entidades que pertencem ao conjunto de entidades EMPREGADO, significando que toda entidade, que é um membro de um desses subgrupos, é também um empregado. Chamamos cada um desses subgrupos uma **subclasse** do tipo entidade EMPREGADO, e o tipo entidade EMPREGADO é conhecido como **superclasse** para cada uma dessas subclasses. A Figura 4.1 mostra como representar diagramaticamente esses conceitos nos diagramas EER.

Chamamos o relacionamento entre uma superclasse e qualquer uma de suas subclasses **relacionamento superclasse/subclasse** ou simplesmente **classe/subclasse**. Em nosso exemplo anterior, EMPREGADO/SECRETÁRIA e EMPREGADO/TÉCNICO são dois relacionamentos classe/subclasse. Observe que uma entidade membro de uma subclasse representa a *mesma entidade do mundo real*, como qualquer membro da superclasse; por exemplo, uma entidade SECRETÁRIA 'Joan Logano' é também o EMPREGADO 'Joan Logano'. Por isso, o membro da subclasse é a mesma entidade da superclasse, mas em um *papel específico*, distinto. Entretanto, quando implementamos um relacionamento superclasse/subclasse em um sistema de banco de dados, podemos representar um membro de uma subclasse como um objeto distinto no banco de dados — digamos, outro registro relacionado à sua entidade superclasse pelo atributo-chave. Na Seção 7.2 discutiremos as várias opções para a representação de relacionamentos superclasse/subclasse em bancos de dados relacionais.

Uma entidade não pode existir em um banco de dados apenas por ser membro de uma subclasse; ela precisa ser também membro de uma superclasse. Assim, uma entidade pode ser incluída opcionalmente como membro de várias subclasses. Por exemplo, um empregado assalariado, que também é um engenheiro, pertence às duas subclasses, ENGENHEIRO e EMPREGADO_ASSALARIADO, do tipo entidade EMPREGADO. Entretanto, não é necessário que toda entidade de uma superclasse seja também membro de alguma subclasse.

Um importante conceito associado a subclasses é o de **tipo de herança**. Recordemos que o *tipo* de uma entidade é definido pelos atributos que ela possui e pelos tipos relacionamento dos quais participa. Como uma entidade de uma subclasse representa a mesma entidade do mundo real da superclasse, ela deve possuir valores para seus atributos específicos, *bem como* valores para seus atributos como membro da superclasse. Dizemos que uma entidade, que é um membro de uma subclasse, **herda** todos os atributos da entidade como um membro da superclasse. A entidade também herda todos os relacionamentos dos quais a superclasse participa. Note que uma subclasse, com seus próprios atributos e relacionamentos específicos (ou locais), e com todos os atributos e relacionamentos que ela herda da superclasse, pode ser considerada um *tipo entidade* em seu senso estrito.

4.2 ESPECIALIZAÇÃO E GENERALIZAÇÃO

4.2.1 Especialização

A **especialização** é o processo de definir um *conjunto de subclasses* de um tipo entidade; esse tipo entidade é chamado **superclasse** da especialização. O conjunto de subclasses que forma uma especialização é definido com base em algumas características de distinção das entidades da superclasse. Por exemplo, o conjunto de subclasses {SECRETÁRIA, ENGENHEIRO, TÉCNICO} é uma especialização da superclasse EMPREGADO, que distingue cada entidade empregado com base em seu *tipo de trabalho*. Podemos ter diversas especializações para o mesmo tipo entidade, baseadas nas diferentes características que as distinguem. Por exemplo, outra especialização do tipo entidade EMPREGADO pode produzir o conjunto de subclasses {EMPREGADO_ASSALARIADO, EMPREGADO_HORISTA}; essa especialização distingue os empregados com base na *forma de pagamento*.

A Figura 4-1 mostra como representamos, diagramaticamente, uma especialização em um diagrama EER. As subclasses que definem uma especialização estão ligadas por linhas a um círculo que representa a especialização, o qual está conectado à superclasse. O *símbolo do subconjunto* em cada linha conectando uma subclasse ao círculo indica a direção do relacionamento superclasse/subclasse. Os atributos que se aplicam apenas às entidades de uma subclasse em particular — tais como VelocidadeDigitacao de SECRETÁRIA — são ligados ao retângulo que representa essa subclasse. Esses são chamados **atributos específicos** (ou **atributos locais**) da subclasse. Similarmente, uma subclasse pode participar de **tipos relacionamento específicos**, como a subclasse EMPREGADO_HORISTA, participante do relacionamento PERTENCE_A da Figura 4.1. Em breve explicaremos o símbolo **d** nos círculos da Figura 4-1 e notações adicionais do diagrama EER.

A Figura 4-2 mostra algumas instâncias de entidade que pertencem às subclasses de especialização {SECRETÁRIA, ENGENHEIRO, TÉCNICO}. Novamente, observe que uma entidade que pertence a uma subclasse representa a *mesma entidade do mundo real* que a entidade conectada a ela na superclasse EMPREGADO. Embora idêntica, a mesma variável é mostrada duas vezes; por exemplo, na

3 Um relacionamento classe/subclasse é freqüentemente chamado **relacionamento IS-A** (ou **IS-na - E-UM**) por causa da forma como nos referimos ao conceito. Dizemos "uma SECRETÁRIA é um EMPREGADO", "um TÉCNICO é um EMPREGADO", e assim por diante.

4 Em algumas linguagens de programação orientadas a objeto, uma restrição comum é que uma entidade (ou objeto) tenha *apenas um tipo*. Em geral, isso é muito restritivo para a modelagem conceitual de banco de dados.

5 Existem muitas notações alternativas para a especialização. Na Seção 4.6 apresentaremos a notação UML, e no Apêndice A, outras notações propostas.

4.2 Especialização e Generalização 63

Figura 4-2, e_x é mostrada tanto em EMPREGADO quanto em SECRETÁRIA. Como a figura sugere, o relacionamento superclasse/subclasse entre EMPREGADO/SECRETÁRIA deve ser um relacionamento 1:1 *em nível de instância* (Figura 3.12).

FIGURA 4.2 Instâncias de uma especialização.

A principal diferença é que, em um relacionamento 1:1, *duas entidades distintas* estão relacionadas, enquanto em um relacionamento superclasse/subclasse, a entidade da subclasse é a mesma do mundo real que a entidade da superclasse, mas desempenhando um *papel especializado* — por exemplo, um EMPREGADO especializado no papel de SECRETÁRIA OU um EMPREGADO especializado no papel de TÉCNICO.

Há, principalmente, duas razões para incluir relacionamentos classe/subclasse e especializações em um modelo de dados. A primeira é que certos atributos podem ser usados em algumas, mas não em todas as entidades da superclasse. Uma subclasse é definida de forma a agrupar as entidades para as quais esses atributos se aplicam. Os membros da subclasse podem, ainda, compartilhar a maioria de seus atributos com outros membros da superclasse. Por exemplo, na Figura 4.1, a subclasse SECRETÁRIA tem o atributo específico VelocidadeDigitacao, ao passo que a subclasse ENGENHEIRO possui o atributo específico TipoEng, mas SECRETÁRIA e ENGENHEIRO compartilham outros atributos herdados do tipo entidade EMPREGADO.

A segunda razão para usar as subclasses é que apenas as entidades que sejam membros de alguma subclasse possam participar de algum tipo relacionamento. Por exemplo, se apenas os EMPREGADOS_HORISTAS podem pertencer a um sindicato, podemos representar esse fato criando a subclasse EMPREGADO_HORISTA de EMPREGADO, e relacionar a subclasse a um tipo entidade SINDICATO, por meio do tipo relacionamento PERTENCE_A, conforme ilustrado na Figura 4-1.

Em resumo, o processo de especialização nos permite fazer o seguinte:

- Definir um conjunto de subclasses de um tipo entidade.
- Estabelecer atributos específicos adicionais para cada subclasse.
 - Estabelecer tipos relacionamento adicionais específicos entre cada subclasse e outros tipos entidade, ou outras subclasses.

4.2.2 Generalização

Podemos pensar em um *processo contrário* de abstração, no qual suprimimos as diferenças entre os diversos tipos entidade, identificamos suas características comuns e os **generalizamos** em uma única **superclasse**, da qual os tipos entidade originais são subclasses especiais. Por exemplo, considere os tipos entidade CARRO e CAMINHÃO da Figura 4-3a. Como eles têm diversos atributos comuns, podem ser generalizados em um tipo entidade VEICULO, como mostrado na Figura 4.3b. Ambas, CARRO e CAMINHÃO, são novas subclasses da **superclasse generalizada** VEICULO. Usamos o termo **generalização** para nos referir ao processo de definição de um tipo entidade generalizada a partir de tipos entidade fornecidos.

Observe que o processo de generalização pode ser visto como, funcionalmente, o inverso do processo de especialização. Portanto, na Figura 4.3 podemos visualizar {CARRO, CAMINHÃO} como uma especialização de VEICULO, em vez de visualizar VEICULO como uma generalização de CARRO e CAMINHÃO. Da mesma forma, na Figura 4-1 podemos ver EMPREGADO como uma generalização de SECRETÁRIA, TÉCNICO e ENGENHEIRO. Uma notação diagramática para a distinção entre generalização e especialização é usada em algumas metodologias de projeto. Uma seta apontando para a superclasse generalizada representa uma generalização, enquanto as setas apontando para as subclasses especializadas representam uma especialização. *Não* usaremos essa notação porque a decisão de qual processo é mais apropriado em uma situação em particular é freqüentemente subjetiva. O Apêndice A traz algumas das notações diagramáticas alternativas, sugeridas para os diagramas de esquema e os diagramas de classe.

FIGURA 4.3 Generalização, (a) Dois tipos entidade, CARRO e CAMINHÃO, (b) Generalizando CARRO e CAMINHÃO na superclasse VEICULO.

Até aqui introduzimos os conceitos de subclasses e relacionamentos superclasse/subclasse tanto para o processo de especialização quanto para o de generalização. Em geral, uma superclasse ou subclasse representa uma coleção de entidades de mesmo tipo e, por isso, também descreve um *tipo entidade*; essa é a razão para as superclasses e subclasses serem mostradas em retângulos, nos diagramas EER, como tipos entidade. Discutiremos, a seguir, as propriedades de especializações e generalizações em mais detalhes.

4.3 CARACTERÍSTICAS E RESTRIÇÕES DA ESPECIALIZAÇÃO E GENERALIZAÇÃO

Discutiremos, primeiro, as restrições que se aplicam à especialização e generalização simples. Por brevidade, nossa discussão refere-se apenas à especialização, ainda que se aplique a ambas, especialização e generalização. Discutiremos, então, as diferenças entre os *reticulados* (*herança múltipla*) de especialização/generalização e as *hierarquias* (*herança simples*), e elaboraremos as diferenças dos processos de especialização e generalização durante o projeto do esquema conceitual do banco de dados.

4.3 Características e Restrições da Especialização e Generalização 65

4.3.1 Restrições na Especialização e Generalização

Em geral, podemos ter diversas especializações definidas para o mesmo tipo entidade (ou superclasse), como mostrado na Figura 4-1. Nesse caso, as entidades podem pertencer a subclasses de cada uma das especializações. Entretanto, uma especialização também pode ter apenas uma *única* subclasse, como a especialização {GERENTE} da Figura 4-1; nesse caso, não usamos a notação de círculo. Em algumas especializações podemos determinar exatamente as entidades que vão se tornar membros de cada subclasse, colocando uma condição no valor de algum atributo da superclasse. Essas subclasses são chamadas **subclasses definidas por predicado** (ou **definidas por condição**). Por exemplo, se o tipo entidade EMPREGADO tiver um atributo TipoTrabalho, como mostrado na Figura 4.4, podemos especificar a condição para pertencer à subclasse SECRETÁRIA por (TipoTrabalho = 'SECRETÁRIA'), que chamamos **definição por predicado** da subclasse. Essa condição é uma restrição que especifica que exatamente as entidades do tipo entidade EMPREGADO, cujo valor do atributo para TipoTrabalho seja 'Secretária', pertencem à subclasse. Exibimos uma subclasse definida por predicado escrevendo a condição do predicado próximo à linha que conecta a subclasse ao círculo de especialização.

Se todas as subclasses de uma especialização têm sua condição determinada pelo *mesmo* atributo da superclasse, a especialização é chamada **especialização definida por atributo**, e o atributo é conhecido como **atributo de definição** da especialização. Exibimos uma especialização definida por atributo colocando o nome do atributo de definição próximo ao arco do círculo para a superclasse, como mostrado na Figura 4.4.

EMPREGADO

TipoTrabalho

VelocidadeDigitacao)/ T . . ,

SECRETARIA

TÉCNICO

ENGENHEIRO

FIGURA 4.4 Notação do diagrama EER para uma especialização definida por atributo em TipoTrabalho.

Quando não temos uma condição que determine que a entidade seja membro de uma subclasse, a subclasse é chamada **definida pelo usuário**. O membro nessa subclasse é determinado pelos usuários do banco de dados na operação que adicionar uma entidade à subclasse, portanto, um membro é *especificado individualmente para cada entidade pelo usuário*, e não por qualquer condição que possa ser avaliada automaticamente.

Outras duas restrições podem ser aplicadas a uma especialização. A primeira é a **restrição de disjunção** (*disjointness*), especificando que as subclasses da especialização devem ser mutuamente exclusivas. Isso significa que uma entidade pode ser membro de, *no máximo, uma* das subclasses da especialização. Uma especialização definida por atributo implica restrição de disjunção se o atributo usado para definir o predicado do associado for monovalorado. A Figura 4.4 ilustra esse caso, em que o **d** no círculo significa *disjunção*. Usamos também a notação **d** para especificar que as subclasses definidas pelo usuário de uma especialização estão sob a restrição de disjunção, como ilustrado pela especialização {EMPREGADO_HORISTA, EMPREGADO, ASSALARIADO} na Figura 4.1. Se as subclasses não estão condicionadas a ser disjuntas, seus conjuntos de entidades podem **sobrepor-se** (overlap),

6

Na terminologia UML, esse atributo é chamado *discriminador*.

66 Capítulo 4 Modelagem com Entidade-Relacionamento Estendido e UML

ou seja, uma mesma entidade (mundo real) pode ser membro de mais de uma subclasse da especialização. Esse caso, que é o padrão, é exibido colocando-se um **o** no círculo, como mostrado na Figura 4.5.

A segunda restrição à especialização é chamada **restrição de integralidade**, que pode ser total ou parcial. Uma restrição de **especialização total** especifica que *toda* entidade na superclasse deve ser um membro de pelo menos uma das subclasses na especialização. Por exemplo, se todo EMPREGADO pode ser um EMPREGADO_HORISTA ou um EMPREGADO_ASSALARIADO, então a especialização (EMPREGADO_HORISTA, EMPREGADO_ASSALARIADO) da Figura 4.1 é uma especialização total de EMPREGADO. Isso é mostrado nos diagramas EER usando-se uma linha dupla para conectar a superclasse ao círculo. Uma linha única é usada para exibir uma **especialização parcial**, admitindo-se que uma entidade não pertença a nenhuma das subclasses. Por exemplo, se algumas entidades EMPREGADO não pertencerem a nenhuma das subclasses {SECRETÁRIA, ENGENHEIRO, TÉCNICO} das figuras 4-1 e 4-4, então a especialização será parcial.

Descrição

PECA

PECA_FABRICADA

PECA_FORNECIDA

FIGURA 4.5 Notação de diagrama EER para uma especialização sobreposta (não disjunta).

Observe que as restrições de disjunção e de integralidade são *independentes*. Por isso, temos as quatro seguintes restrições possíveis na especialização:

- Disjunção total
- Disjunção parcial
- Sobreposição total
- Sobreposição parcial

É claro que a restrição correta é determinada pelo sentido, no mundo real, que se aplica a cada especialização. Uma superclasse identificada por meio do processo de *generalização* é, em geral, **total**, porque a superclasse é *derivada de* subclasses e, por isso, contém apenas as entidades que estão nas subclasses.

Certas regras de inserção e exclusão aplicam-se à especialização (e generalização), como consequência das restrições especificadas no início. Algumas dessas regras são as seguintes:

- Deletar uma entidade de uma superclasse implica que ela seja excluída, automaticamente, de todas as subclasses às quais pertence.
- Inserir uma entidade em uma superclasse implica que ela seja inserida, obrigatoriamente, em todas as subclasses *definidas por predicado* (ou *definidas por atributo*) para as quais a entidade satisfizer a definição por predicado.
- Inserir uma entidade em uma superclasse de *especialização total* implica que a entidade seja, obrigatoriamente, inserida em pelo menos uma das subclasses da especialização.

O leitor é encorajado a fazer uma lista de regras completa para as inserções e exclusões nos vários tipos de especializações.

7 A notação de uso de linhas únicas ou duplas é similar à usada para a participação parcial ou total de um tipo entidade em um tipo relacionamento, como descrito no Capítulo 3.

4.3 Características e Restrições da Especialização e Generalização 67

4.3.2 Hierarquias e Reticulados de Especialização e Generalização

Uma subclasse pode ter, ela mesma, subclasses próprias, formando uma hierarquia ou um reticulado de especializações. Por exemplo, na Figura 4.6, ENGENHEIRO é uma subclasse de EMPREGADO, e é também uma superclasse de GERENTE_ENGENHARIA; esta é uma restrição do mundo real, que o gerente de engenharia deva ser um engenheiro. Uma **hierarquia de especialização** restringe todas as suas *subclasses* a participar de *apenas um* relacionamento classe/subclasse, isto é, cada subclasse tem apenas um pai, o que resulta em uma estrutura de árvore. Já para uma **especialização reticulada**, uma subclasse pode participar de *mais de um* relacionamento classe/subclasse. Portanto, a Figura 4.6 é um reticulado.

A Figura 4.7 mostra outra especialização reticulada de mais de um nível. Ela pode ser parte do esquema conceitual do banco de dados UNIVERSIDADE. Note que essa disposição seria uma hierarquia não fosse pela subclasse ASSISTENTE_ALUNO, que é subclasse em dois relacionamentos classe/subclasse distintos. Na Figura 4.7, todas as entidades pessoa representadas no banco de dados são membros do tipo entidade PESSOA, que está especializada nas subclasses (EMPREGADO, EXALUNO, ALUNO). Essa especialização é de sobreposição; por exemplo, um ex-aluno pode também ser um empregado, e também pode ser um aluno prosseguindo seus estudos em um grau mais avançado. A subclasse ALUNO corresponde a uma superclasse para a especialização {ALUNO_GRADUADO, ALUNO_NAÓGRADUADO}, enquanto o EMPREGADO pertence a uma superclasse para a especialização {ASSISTENTE_ALUNO, DOCENTE, AUXILIAR}. Observe que um ASSISTENTE_ALUNO é também uma subclasse de ALUNO. Finalmente, ASSISTENTE_ALUNO é superclasse para a especialização {ASSISTENTE_PESQUISA, ASSISTENTE_ENSINO}.

EMPREGADO

SECRETARIA

ENGENHEIRO GERENTE

u

EMPREGADO_HORISTA

u

EMPREGADO_ASSALARIADO

GERENTE_ENGENHARIA

FIGURA 4.6 Um reticulado de especialização, com a subclasse GERENTE_ENGENHARIA compartilhada.

Em uma especialização reticulada ou hierárquica, uma subclasse herda os atributos não somente de sua superclasse direta, como também de todas as superclasses predecessoras, *até a raiz* da hierarquia ou reticulado. Por exemplo, uma entidade em ALUNO_GRADUADO herda todos os atributos dessa entidade como um ALUNO ou uma PESSOA. Note que uma entidade pode existir em diversos *nodo folha* da hierarquia, no qual um **nodo folha** é uma classe que *não possui subclasses próprias*. Por exemplo, um membro de ALUNO_GRADUADO também pode ser um membro de ASSISTENTE_PESQUISA.

Uma subclasse com *mais de uma* superclasse é chamada uma **subclasse compartilhada**, como GERENTE_ENGENHARIA na Figura 4.6. Isso leva ao conceito conhecido por **herança múltipla**, em que a subclasse compartilhada GERENTE_ENGENHARIA herda, diretamente, os atributos e relacionamentos de múltiplas classes. Observe que a existência de, pelo menos, uma subclasse compartilhada, leva a um reticulado (e, portanto, a uma *herança múltipla*); se não existissem subclasses compartilhadas, teríamos uma hierarquia, em vez de um reticulado. Uma importante regra relacionada à herança múltipla pode ser ilustrada pelo exemplo da subclasse compartilhada ASSISTENTE_ALUNO na Figura 4.7, com atributos herdados de EMPREGADO e ALUNO. Aqui, ambos, EMPREGADO e ALUNO, herdam *os mesmos atributos* de PESSOA. As regras declaram que, se um atributo (ou relacionamento) originado na *mesma superclasse* (PESSOA) é herdado mais de uma vez por diferentes caminhos (EMPREGADO e ALUNO) no reticulado, então deveria ser incluído apenas uma vez na subclasse compartilhada (ASSISTENTE_ALUNO). Portanto, os atributos de PESSOA são herdados *apenas uma vez* na subclasse ASSISTENTE_ALUNO da Figura 4.7.

68 Capítulo 4 Modelagem com Entidade-Relacionamento Estendido e UML

```

(sexo) Nome \
PESSOA
C^DeptHarT^
AUXILIAR
DOCENTE
ASSISTENTE_ALUNO
ALUNO_GRADUADO
ALUNO_NAOGRADUADO
ProgramaTitulacao
ASSISTENTE_PESQUISA
ASSISTENTE_ENSINO

```

FIGURA 4.7 Um reticulado de especialização, com herança múltipla, para um banco de dados UNIVERSIDADE.

É importante observar aqui que alguns modelos e linguagens *não permitem* herança múltipla (subclasses compartilhadas). Nesses modelos, é necessário criar subclasses adicionais para cobrir todas as combinações possíveis de classes que abriguem as entidades pertencentes simultaneamente a várias dessas classes. Portanto, qualquer especialização de *sobreposição* poderá exigir diversas subclasses adicionais. Por exemplo, na especialização de sobreposição PESSOA em {EMPREGADO, EXALUNO, ALUNO} (ou {E, X, A} para abreviar), será necessário criar sete subclasses para PESSOA, de forma a cobrir todos os possíveis tipos entidades: E, X, A, E_X, E_A, X_A e E_X_A. Obviamente, isso acaba por elevar o grau de complexidade.

Também é importante notar que alguns mecanismos de herança múltipla não permitem que uma entidade tenha diversos tipos e, assim, a entidade poderá ser membro *de apenas uma classe*. Nesses modelos, também serão necessárias subclasses adicionais compartilhadas, como nodos folha, para cobrir todas as combinações possíveis de classes que possam ter entidades pertencentes, simultaneamente, a todas essas classes. Portanto, poderíamos ter as mesmas sete subclasses de PESSOA.

8 Em alguns modelos, a classe se torna posteriormente restrita a um *nodo folha* em uma hierarquia ou reticulado.

4.4 Modelagem de Tipos União Usando as Categorias 69

Embora tenhamos usado especialização para ilustrar a nossa discussão, os conceitos similares *se aplicam, igualmente*, à generalização, como mencionado no início desta seção. Portanto, podemos falar, também, de **hierarquias de generalização** e **reticulados de generalização**.

4.3.3 Utilizando a Especialização e Generalização para o Refinamento de Esquemas Conceituais

Elaboramos, agora, as diferenças entre os processos de especialização e generalização, e como elas são usadas para refinamento de esquemas conceituais durante o projeto do banco de dados conceitual. No processo de especialização iniciamos, em geral, com um tipo entidade, então definimos as subclasses desse tipo entidade por sucessivas especializações, isto é, definimos repetidamente mais agrupamentos específicos do tipo entidade. Por exemplo, quando projetamos o reticulado de especialização da Figura 4.7, primeiro especificamos um tipo entidade PESSOA para o banco de dados universidade. Então, descobrimos que três tipos de pessoas deveriam ser representadas no banco de dados: os empregados da universidade, os ex-alunos e os alunos. Criamos a especialização {EMPREGADO, EXALUNO, ALUNO} para esse propósito, e escolhemos a restrição de sobreposição porque uma pessoa pode pertencer a mais de uma das subclasses. Então, adiante, especializamos EMPREGADO em {AUXILIAR, DOCENTE, ASSISTENTE_ALUNO}, e especializamos ALUNO em {ALUNO_GRADUADO, ALUNO_NAOGRAUADO}. Finalmente, especializamos ASSISTENTE_ALUNO em {ASSISTENTE_PESQUISA, ASSISTENTE_ENSINO}. Essa especialização sucessiva corresponde ao **processo de refinamento conceitual top-down** (de cima para baixo) durante o projeto do esquema conceitual. Até aqui, construímos uma hierarquia; percebemos, então, que ASSISTENTE_ALUNO é uma subclasse compartilhada, uma vez que ela também é uma subclasse de ALUNO levando a um reticulado.

É possível chegar à mesma hierarquia ou reticulado por outra direção. Nesse caso, o processo implica generalização em vez de especialização e corresponde a uma **síntese conceitual bottom-up** (de baixo para cima). Nesse caso, os projetistas devem, primeiro, descobrir os tipos entidade, como AUXILIAR, DOCENTE, EXALUNOS, ALUNO_GRADUADO, ALUNO_NAOGRAUADO, ASSISTENTE_PESQUISA, ASSISTENTE_ENSINO, e assim por diante, então eles generalizam {ALUNO_GRADUADO, ALUNO_NAOGRAUADO} em ALUNO; depois generalizam {ASSISTENTE_PESQUISA, ASSISTENTE_ENSINO} em ASSISTENTE_ALUNO; depois generalizam {AUXILIAR, DOCENTE, ASSISTENTE_ALUNO} em EMPREGADO; e finalmente generalizam {EMPREGADO, EXALUNO, ALUNO} em PESSOA.

Em termos estruturais, as hierarquias ou os reticulados resultantes de qualquer um dos processos podem ser idênticos; a única diferença refere-se à maneira ou à ordem na qual as superclasses e subclasses do esquema foram identificadas. Na prática, é provável que nem o processo de generalização nem o processo de especialização seja rigorosamente seguido, mas empregada uma combinação dos dois processos. Nesse caso, novas classes são continuamente incorporadas à hierarquia ou ao reticulado assim que se tornam visíveis aos usuários e projetistas. Observe que a noção de representação de dado e conhecimento pelo uso de hierarquias e reticulados de superclasse/subclasse é bastante comum em sistemas baseados em conhecimento e em sistemas especialistas, que combinam a tecnologia de banco de dados com as técnicas de inteligência artificial. Por exemplo, os esquemas de representação de conhecimento baseado *emframes* parecem-se, fortemente, com as hierarquias de classe. A especialização também é comum nas metodologias de projeto de engenharia de software que são baseadas no paradigma da orientação a objeto.

4.4 MODELAGEM DE TIPOS UNIÃO USANDO AS CATEGORIAS

Todos os relacionamentos de superclasse/subclasse que vimos até aqui têm uma *superclasse única*. Uma subclasse compartilhada, como GERENTE_ENGENHARIA, no reticulado da Figura 4.6, corresponde a uma subclasse nos três *diferentes* relacionamentos de superclasse/subclasse, em que cada um dos três tem uma superclasse *única*. Não é incomum, entretanto, que surja a necessidade de modelar um único relacionamento superclasse/subclasse com *mais de uma* superclasse, na qual a superclasse represente os diferentes tipos entidade. Nesse caso, a subclasse representará uma coleção de objetos que é um subconjunto da UNIÃO de diferentes tipos entidade; chamamos essa *subclasse tipo união* ou **categoria**.

Por exemplo, suponha que tenhamos três tipos de entidade: PESSOA, BANCO e EMPRESA. Em um banco de dados para o registro de veículos, um proprietário de um veículo pode ser uma pessoa, um banco (que possui o direito de alienação de um veículo) ou uma empresa. Precisamos criar uma classe (coleção de entidades) que inclua entidades de todos os três tipos, para desempenharem o papel de *proprietário do veículo*. Uma categoria PROPRIETÁRIO, que é uma *subclasse da UNIÃO* dos três conjuntos de

9 Nosso uso do termo *categoria* é baseado no modelo ECR (*Entity-Category-Relationship* — Relacionamento Entidade-Categoria) (Elmas-rietaí., 1985).

entidade de EMPRESA, BANCO e PESSOA, é criada com esse propósito. Exibimos as categorias em um diagrama EER, conforme mostrado na Figura 4.8. As superclasses EMPRESA, BANCO e PESSOA estão conectadas a um círculo com o símbolo U, que representa a *operação de união de conjunto*. Um arco com o símbolo de subconjunto conecta o círculo à (subclasse) categoria PROPRIETÁRIO. Se for necessária uma definição por predicado, ela será exibida próximo à linha da superclasse à qual o predicado se aplica.

FIGURA 4.8 Duas categorias (tipos união): PROPRIETÁRIO e VEICULO_REGISTRADO.

Na Figura 4.8 temos duas categorias: PROPRIETÁRIO, que é uma subclasse da união de PESSOA, BANCO e EMPRESA, e VEICULO_REGISTRADO, que é uma subclasse da união de CARRO e CAMINHÃO.

Uma categoria possui duas ou mais superclasses que podem representar *diferentes tipos entidade*, enquanto outros relacionamentos de superclasse/subclasse sempre têm uma superclasse única. Podemos comparar uma categoria, como PROPRIETÁRIO da Figura 4.8, com a subclasse compartilhada GERENTE_ENGENHARIA da Figura 4.6. Essa última é uma subclasse *de cada uma* das três superclasses ENGENHEIRO, GERENTE e EMPREGADO_ASSALARIADO, assim, uma entidade que é membro de GERENTE_ENGENHARIA deve existir em *todas as três*. Isso representa a restrição que um gerente de engenharia deva ser um ENGENHEIRO, um GERENTE e um EMPREGADO_ASSALARIADO, isto é, GERENTE_ENGENHARIA é um subconjunto da *interseção* das três subclasses (conjuntos de entidades). Porém, uma categoria é um subconjunto

da *união* dessas superclasses. Daí uma entidade que seja membro de PROPRIETÁRIO deve existir em *apenas uma* das superclasses. Isso representa, na Figura 4.8, a restrição de que PROPRIETÁRIO precisa ser uma EMPRESA, um BANCO ou uma PESSOA.

O atributo de herança trabalha mais seletivamente quando aplicado às categorias. Por exemplo, na Figura 4.8, cada entidade PROPRIETÁRIO herda os atributos da EMPRESA, da PESSOA ou do BANCO, dependendo da superclasse à qual a entidade pertence. Entretanto, uma subclasse compartilhada, como GERENTE_ENGENHARIA (Figura 4.6), herda *todos* os atributos de suas superclasses EMPREGADO_ASSALARIADO, ENGENHEIRO e GERENTE.

É interessante observar a diferença entre a categoria VEICULO_REGISTRADO (Figura 4.8) e a superclasse generalizada VEICULO (Figura 4.3b). Na Figura 4.3b, todo carro e todo caminhão referem-se a um VEICULO mas na Figura 4-8 a categoria VEICULO_REGISTRADO inclui alguns carros e alguns caminhões, mas não necessariamente todos eles (por exemplo, alguns carros ou caminhões podem não estar registrados). Em geral, as especializações ou generalizações como as da Figura 4.3b, se fossem *parciais*, não excluiriam os outros tipos entidade VEICULO, como motocicletas. Entretanto, uma categoria como VEICULO_REGISTRADO, na Figura 4.8, implica que apenas carros e caminhões, e não outros tipos de entidades, possam ser membros de VEICULO_REGISTRADO.

Uma categoria pode ser total ou parcial. Uma categoria total controla a *união* de todas as entidades em suas superclasses, ao passo que uma categoria parcial pode controlar um *subconjunto da união*. Uma categoria total é representada por uma linha dupla conectando a categoria ao círculo, enquanto as categorias parciais são indicadas por uma linha simples.

As superclasses de uma categoria podem ter diferentes atributos-chave, como demonstrado pela categoria PROPRIETÁRIO da Figura 4.8, ou elas podem ter o mesmo atributo-chave, como demonstrado pela categoria VEICULO_REGISTRADO. Note que se uma categoria é total (não parcial), ela pode ser representada, alternativamente, como uma especialização total (ou uma generalização total). Nesse caso, a escolha sobre qual representação usar é subjetiva. Se as duas classes representarem o mesmo tipo entidade e compartilharem numerosos atributos, incluindo os mesmos atributos-chave, a especialização/generalização é preferida; de outra maneira, a categorização (tipo união) é mais apropriada.

4.5 UM EXEMPLO — UNIVERSIDADE — DE ESQUEMA EER E DEFINIÇÕES FORMAIS PARA O MODELO EER

Nesta seção daremos primeiro um exemplo de um esquema de banco de dados no modelo EER para ilustrar o uso de vários conceitos discutidos aqui e no Capítulo 3. A seguir, resumiremos os conceitos do modelo EER e os definiremos formalmente, da mesma maneira que fizemos com os conceitos básicos do modelo ER no Capítulo 3.

4.5.1 O Exemplo de Banco de Dados UNIVERSIDADE

Na nossa aplicação de um banco de dados exemplo, consideramos um banco de dados UNIVERSIDADE que controle seus alunos, programas, históricos e registros, bem como os cursos oferecidos. O banco de dados também controla os projetos de pesquisa patrocinados para o corpo docente e alunos de graduação. Esse esquema é mostrado na Figura 4.9. Segue uma discussão dos requisitos que levaram a esse esquema.

Para cada pessoa o banco de dados mantém informação do Nome da pessoa [Nome], número do seguro social [SSN], endereço [Endereço], sexo [Sexo] e data de nascimento [DNasc]. Duas subclasses do tipo entidade PESSOA foram identificadas: DOCENTE e ALUNO. Os atributos específicos de DOCENTE são categoria [Categoria] (assistente, associado, adjunto, pesquisador, visitante etc), escritório [FEscritório], telefone do escritório [FFone] e salário [Salário]. Todos os docentes estão relacionados ao(s) departamento(s) acadêmico(s) aos quais são afiliados [PERTENCE] (um docente pode ser associado a diversos departamentos, assim o relacionamento é M:N). Um atributo específico de ALUNO é [Turma] (calouro = 1, segundalista = 2, ..., aluno graduado = 5). Cada aluno também está relacionado a um departamento de habilitação e a um opcional, se houver algum ([HABILITA] e [OPTA]), para as disciplinas em que está matriculado atualmente [REGISTRADO], e a cursos completados [HISTÓRICO]. Cada instância de histórico inclui a nota recebida pelo aluno [Nota] na disciplina do curso.

GRAD_ALUNO é uma subclasse de ALUNO com a definição por predicado Turma = 5. Para cada aluno graduado mantemos uma lista dos títulos acadêmicos anteriores em um atributo multivalorado composto [Formação]. Relacionamos também o aluno graduado a um orientador docente [ORIENTADOR] e a um comitê de tese

[BANCA], se existir algum.

Um departamento acadêmico tem os atributos nome [NomeDepto], telefone [FoneDepto] e número do escritório [Escritório] e está relacionado ao docente que é seu chefe [CHEFIA] e à faculdade à qual ele pertence [FDEPTO]. Cada faculdade tem os atributos nome [FNome], número do escritório [FEscritorio] e o nome de seu reitor [Reitor].

Um curso tem os atributos número do curso [C#], nome do curso [CNome] e descrição do curso [CDesc]. Diversas disciplinas de cada curso são oferecidas, cada qual com os atributos número [Disc#] e o ano e o trimestre nos quais a disciplina foi

72 Capítulo 4 Modelagem com Entidade-Relacionamento Estendido e UML

oferecida ([Ano] e [Trim]). Números únicos identificam cada disciplina. As disciplinas que estão sendo oferecidas durante o trimestre corrente estão em uma subclasse de DISCIPLINA, DISCIPLINA_CORRENTE, com a definição por predicado Trimestre = Trim-Corrente e Ano = AnoCorrente. Cada disciplina está relacionada ao instrutor que a ministrou ou está ministrando ([ENSINAR]), se esse instrutor estiver no banco de dados.

FIGURA 4.9 Um esquema EER conceitual para um banco de dados UNIVERSIDADE.

A categoria INSTRUTOR_PESQUISADOR é um subconjunto da união de DOCENTE e GRAD_ALUNO e inclui todos os docentes e alunos graduados contratados para o ensino ou pesquisa. Finalmente, o tipo entidade BOLSA mantém o controle das bolsas de pesquisa e dos contratos financiados para a universidade. Cada bolsa tem os atributos título da bolsa [Titulo], número da bolsa [Num], a agência financiadora [Agencia] e a data de início [DataIn]. Uma bolsa está relacionada a um investigador principal [IP] e a todos

10 Pressupomos que o sistema *trimestral* seja mais usado nessa universidade que o *semestral*.

4.6 Representação da Especialização/Generalização e Herança em Diagramas de Classe UML 73

os pesquisadores subsidiados por ela [SUBSÍDIO]. Cada instância de subsídio tem como atributos a data de início do subsídio [Início], a data final do subsídio (se conhecida) [Fim] e a porcentagem de tempo gasta no projeto [Tempo] pelo pesquisador apoiado.

4.5.2 Definições Formais para os Conceitos do Modelo EER

Vamos, agora, resumir os conceitos do modelo EER e fornecer as definições formais. Uma **classe** é um conjunto ou coleção de entidades; isso inclui qualquer construção de esquema EER que agrupe as entidades, como tipos entidade, subclasses, super-classes e categorias. Uma **subclasse** S é uma classe cujas entidades devem sempre ser um subconjunto das entidades de outra classe, chamada **superclasse** C do **relacionamento superclasse/subclasse** (ou **IS-A**). Indicamos esse relacionamento por C/S . Para o relacionamento superclasse/subclasse, precisamos ter sempre s/c .

Uma **especialização** $Z = \{S_1, S_2, \dots, S_n\}$ é um conjunto de subclasses que têm a mesma superclasse G , isto é, G/S , é um relacionamento superclasse/subclasse para $i = 1, 2, \dots, n$. G é chamado **tipo entidade generalizada** (ou **superclasse** da especialização, ou **generalização** de subclasses $\{S_1, S_2, \dots, S_n\}$). Z é o **total** sempre que tivermos (a qualquer momento)

$\bigcup S_i = G$

$i=1$

Do contrário, Z é **parcial**. Z é **disjunto** sempre que tivermos

$S_i \cap S_j = \emptyset$ (conjunto vazio) para $i \neq j$

Do contrário, Z é **sobreposto**.

Uma subclasse S de C é **definida por predicado** se um predicado p nos atributos de C for usado para especificar quais entidades em C são membros de S , isto é, $S = C[p]$, em que $C[p]$ é o conjunto de entidades em C que satisfazem p . Uma subclasse que não é definida por um predicado é chamada **definida por usuário**.

Uma especialização Z (ou generalização G) é **definida por atributo** se um predicado ($A = q$), no qual A for um atributo de G e q um valor constante do domínio de A , for usado para especificar os membros de cada subclasse S_j em Z . Observe que se $C_j \wedge C$: para i/j , e A for um atributo monovalorado, então a especialização será disjunta.

Uma **categoria** T é uma classe que é um subconjunto da união de n superclasses definidas D_1, D_2, \dots, D_n , $n > 1$, e é formalmente especificada como segue:

$T \subseteq (D_1 \cup D_2 \cup \dots \cup D_n)$

Um predicado p_i nos atributos de D_i pode ser usado para especificar os membros de cada D_i , que são os membros de T . Se um predicado for especificado em todo D_i , temos

$T = (D_1[p_1] \cup D_2[p_2] \cup \dots \cup D_n[p_n])$

Deveríamos, agora, estender a definição de **tipo relacionamento** dada no Capítulo 3, permitindo que toda classe — não apenas todo tipo entidade — participe do relacionamento. Por isso deveríamos substituir as palavras tipo *entidade* por *classe*, nessa definição. A notação gráfica EER é compatível com a ER, pois todas as classes são representadas por retângulos.

4.6 REPRESENTAÇÃO DA ESPECIALIZAÇÃO/GENERALIZAÇÃO E HERANÇA EM DIAGRAMAS DE CLASSE UML

Discutiremos agora a notação UML para generalização/especialização e herança. Já apresentamos a notação e terminologia básica de diagrama de classe UML na Seção 3.8. A Figura 4.10 ilustra um possível diagrama de classe UML, correspondente ao diagrama EER da Figura 4.7. A notação básica para generalização é feita pela conexão das subclasses, por linhas verticais, a uma linha horizontal com um triângulo ligando-a por meio de outra linha vertical à superclasse (Figura 4-10). Um triângulo em branco indica uma especialização/generalização com a restrição de *disjunção*, e um triângulo cheio indica uma restrição de

11 O emprego da palavra *classe*, aqui, difere de seu uso mais comum nas linguagens de programação orientadas a objeto, como C++. Em C++, uma classe é uma definição de tipo estruturado na qual podem ser aplicadas as funções (operações).

74 Capítulo 4 Modelagem com Entidade-Relacionamento Estendido e UML

sobreposição. A superclasse raiz é chamada classe base, e os nodos folhas são conhecidos como **classes-folha**. Heranças únicas e múltiplas são permitidas.

A discussão anterior e o exemplo dado (e a Seção 3.8) dão uma breve descrição de diagramas de classe na terminologia UML. Há muitos detalhes que não discutiremos porque estão fora do escopo deste livro, embora relevantes, principalmente para a engenharia de software. Por exemplo, as classes podem ser de vários tipos:

- Classes abstratas definem os atributos e operações, mas não têm objetos correspondentes a essas classes. São usadas, principalmente, para especificar um conjunto de atributos e operações que não podem ser herdados.
- Classes concretas podem ter objetos (entidades) instanciados que pertencem às classes.
- Classes *template* (de molde) especificam moldes que poderão ser usados, mais tarde, na definição de outras classes.

```

PESSOA
Nome
SSN
DataNasc
Sexo
Endereço
idade
EMPREGADO
salário
contrata r_emp
EXALUNO
novo_exaluno
FORMAÇÃO
Ano
Grau
Habilitação
FUNCIONÁRIO
Funcao
contratar_funcionario
DOCENTE
Categoria
promoção
ASSISTENTE_ALUNO
PorcentagemTempo
contratar_aluno
ALUNO_GRADUADO
ProgramaFormacao
ALUNO
Depto_Habilitacao
mudar^habilitacao
mudar_programajornacao
ALUNO_NAOGRADUADO
Turma
mudar_turma
ASSISTENTE_PESQUISA
Projeto
mudarprojeto
ASSISTENTE^ENSINO
Curso
designar_para_curso

```

FIGURA 4.10 Um diagrama de classe correspondente ao diagrama EER da Figura 4.7 ilustrando as notações UML para a especialização/generalização.

Em projeto de banco de dados estamos preocupados, principalmente, com a especificação de classes concretas cujas coleções de objetos estão permanentemente (ou persistentemente) armazenadas no banco de dados. As notas bibliográficas ao final deste capítulo dão algumas referências de livros que descrevem os detalhes completos de UML. O material adicional relacionado à UML encontra-se no Capítulo 12, e a modelagem de objeto em geral é discutida adiante, no Capítulo 20.

4.7 TIPOS RELACIONAMENTO COM GRAU MAIOR QUE DOIS

Na Seção 3.4-2 definimos **grau** de um tipo relacionamento como o número de tipos entidade participantes, e chamamos um tipo relacionamento de grau dois de *binário*, e um tipo relacionamento de grau três de *ternário*. Nesta seção elaboramos as diferenças entre os relacionamentos binários e os de graus superiores, quando escolher os relacionamentos de graus superiores ou binários, e as restrições em relacionamentos de grau superior.

4.7.1 Escolhendo entre Relacionamentos Binários e Ternários (ou de Grau Superior)

A notação de diagrama ER para um tipo relacionamento ternário é mostrada na Figura 4.11a, que exhibe o esquema para o tipo relacionamento FORNECE, que foi exibido em nível de instância na Figura 3.10. Recordemos que o conjunto de relacionamento FORNECE é um conjunto de instâncias de relacionamento (s, j, p) em que s é um FORNECEDOR que supre atualmente um LOTE p para um PROJETO j . Em geral, um tipo relacionamento R , de grau n , terá n retas em um diagrama ER, cada uma conectando R a cada tipo entidade participante.

FIGURA 4.11 Tipos de relacionamentos ternários. (a) O relacionamento FORNECE, (b) Três relacionamentos binários não equivalentes a FORNECE, (c) FORNECE representado como um tipo entidade fraca.

A Figura 4.11b mostra um diagrama ER para os três tipos de relacionamento, PODE_FORNECER, USA e FORNECE. Em geral, um tipo de relacionamento ternário representa informações diferentes das dos três tipos de relacionamento binário. Considere os três tipos de relacionamento binário PODE_FORNECER, USA e FORNECE. Suponha que PODE_FORNECER, entre FORNECEDOR e LOTE, inclua uma instância (s, p) quando fornecedor s *puder fornecer* lote p (a qualquer projeto); USA, entre PROJETO e LOTE, inclua uma instância (j, p) quando projeto j *usar* lote p ; e FORNECE, entre FORNECEDOR e PROJETO inclua uma instância (s, j) quando

fornecedor s *fornecer* algum

76 Capítulo 4 Modelagem com Entidade-Relacionamento Estendido e UML

lote para projeto. A existência de três instâncias de relacionamento (s, p) , (j, p) e (s, j) em PODE_FORNECER, USA e FORNECE, respectivamente, não implica, necessariamente, a existência de uma instância $\{s, j, p\}$ em um relacionamento ternário FORNECE, porque o *sentido é diferente*. Normalmente, é difícil decidir se um relacionamento em particular deveria ser representado como um tipo relacionamento de grau n ou deveria ser quebrado em diversos tipos relacionamento de graus menores. O projetista deve basear sua decisão em semânticas ou no sentido da situação em particular que está sendo representada. A solução típica é incluir o relacionamento ternário acrescido de um ou *mais* relacionamentos binários, se eles representarem diferentes sentidos e se todos forem necessários à aplicação.

Algumas ferramentas de projeto de banco de dados são baseadas em variações do modelo ER, que permitem apenas os relacionamentos binários. Nesse caso, um relacionamento ternário, tal qual FORNECE, deve ser representado como um tipo entidade fraca, sem chave parcial e com três relacionamentos de identificação. Os três tipos entidade participantes, FORNECEDOR, LOTE e PROJETO são, juntos, os tipos entidade forte (Figura 4.11c). Portanto, uma entidade tipo entidade fraca FORNECE, da Figura 4.11c, é identificada pela combinação de suas três entidades fortes: FORNECEDOR, LOTE e PROJETO.

Outro exemplo é mostrado na Figura 4.12. O tipo relacionamento ternário OFERECE representa a informação de instrutores oferecendo cursos durante um semestre em particular, portanto, inclui uma instância de relacionamento (i, s, c) quando INSTRUTOR i oferece CURSO c durante SEMESTRE s . Os três tipos de relacionamento mostrados na Figura 4.12 têm o seguinte significado: PODE_ENSINAR relaciona um curso aos instrutores que *podem ensiná-lo*; ENSINOU_DURANTE relaciona um semestre ao instrutor que *ensinou algum curso* durante aquele semestre, e OFERECIDO_DURANTE relaciona um semestre aos cursos oferecidos durante aquele semestre por *algum instrutor*. Esses relacionamentos ternário e binário representam diferentes informações, mas, certamente, teriam restrições entre os relacionamentos. Por exemplo, uma instância de relacionamento (i, s, c) não deveria existir em OFERECE *a menos* que exista uma instância (i, s) em ENSINOU_DURANTE, uma instância (s, c) em OFERECIDO_DURANTE, e uma instância (i, c) em PODE_ENSINAR. Entretanto, o contrário nem sempre é verdadeiro: podemos ter instâncias (i, s) , (s, c) e (i, c) nos três tipos de relacionamentos binários, sem nenhuma instância correspondente (i, s, c) em OFERECE. Observe que, nesse exemplo, baseados no significado dos relacionamentos, podemos inferir instâncias em ENSINOU_DURANTE e OFERECIDO_DURANTE a partir das instâncias em OFERECE, mas não podemos inferir instâncias de PODE_ENSINAR, portanto, ENSINOU_DURANTE e OFERECIDO_DURANTE são redundantes e poderiam ser omitidos.

CURSO

FIGURA 4.12 Outro exemplo de tipos relacionamento ternário *versus* binário.

Embora, em geral, três relacionamentos binários não possam substituir um relacionamento ternário, eles poderão fazê-lo sob certas restrições *adicionais*. No nosso exemplo, se o relacionamento PODE_ENSINAR for 1:1 (um instrutor pode ensinar um curso, e um curso pode ser ensinado por apenas um instrutor), então o relacionamento ternário OFERECE poderia ser omitido, pois poderia ser inferido dos três relacionamentos binários PODE_ENSINAR, ENSINOU_DURANTE e OFERECIDO_DURANTE. O projetista do esquema deve analisar o sentido de cada situação específica para decidir quais tipos de relacionamentos, binário e ternário, serão necessários.

Observe que é possível ter um tipo entidade fraca como um tipo relacionamento de identificação ternário (ou binário). Nesse caso, o tipo entidade fraca pode ter *diversos* tipos de entidades fortes. Na Figura 4-13 é apresentado um exemplo.

4.7.2 Restrições em Relacionamentos Ternários (ou de Grau Superior)

Há duas notações para a especificação de restrições estruturais em relacionamentos ternário, e elas definem as restrições diferentes. Ambas, portanto, poderiam *ser usadas*, caso seja importante especificar as restrições estruturais em um relacionamento ternário ou de grau superior. A primeira notação é baseada naquela de razão de cardinalidade de relacionamentos binários, mostrada na Figura 3.2. Aqui, 1, ou M ou N será especificado em cada arco de participação (ambos os símbolos, M e N, indicam *muitos* ou *qualquer número*). Vamos ilustrar essa restrição usando o relacionamento FORNECE na Figura 4.11.

Lembre-se de que o conjunto de relacionamento FORNECE é um conjunto das instâncias do relacionamento (s, j, p) , em que s é um FORNECEDOR; j , um PROJETO; e p , um LOTE. Suponha que exista restrição para uma combinação projeto-lote em particular, e que apenas um fornecedor será usado (somente um fornecedor proporciona um lote em particular a um projeto em particular). Nesse caso, colocamos 1 na participação de FORNECEDOR e M, N nas participações de PROJETO e LOTE, como na Figura 4-11. Isso especifica a restrição que uma combinação particular (j, p) pode aparecer *no máximo uma vez* no conjunto de relacionamentos, pois cada uma das combinações (projeto, lote) é determinada por um único fornecedor. Portanto, qualquer instância do relacionamento (s, j, p) é *identificada univocamente* no conjunto de relacionamentos pelas combinações (j, p) , o que faz de (j, p) uma chave para o conjunto de relacionamentos. Nessa notação, as participações que têm a especificação um não são chamadas para fazer parte da chave identificadora do conjunto de relacionamentos.

A segunda notação é baseada na notação (min, max) exibida na Figura 3.15 para os relacionamentos binários. Um (min, max) em uma participação especifica, aqui, que cada entidade está relacionada no mínimo a *min* e, no máximo, a *max* instâncias do relacionamento, no conjunto de relacionamentos. Essas restrições não têm responsabilidade na determinação da chave de um relacionamento binário, no qual $n > 2$, mas especificam um tipo diferente de restrição, que limita o número de instâncias de relacionamento em que cada entidade possa participar.

FIGURA 4.13 Tipo entidade fraca ENTREVISTA com um tipo relacionamento de identificação ternário.

4.8 ABSTRAÇÃO DE DADOS, REPRESENTAÇÃO DO CONHECIMENTO E CONCEITOS DE ONTOLOGIA

Nesta seção discutiremos, em termos abstratos, alguns dos conceitos de modelagem que descrevemos muito especificamente em nossa apresentação dos modelos ER e EER no Capítulo 3 e no início deste capítulo. Essa terminologia é usada tanto na modelagem de dados conceitual como na literatura sobre inteligência artificial, quando se discute a **representação do conhecimento** (abreviado por RC). O objetivo das técnicas de RC é desenvolver os conceitos para a modelagem mais precisa de alguns **domínios do conhecimento** para a criação de uma **ontologia** que descreva os conceitos do domínio. Isso é, então, usado para armazenar e manipular o conhecimento para a concepção de inferências, tomadas de decisões ou apenas para responder a perguntas. Os objetivos da RC são similares àqueles dos modelos de dados semânticos, mas há algumas similaridades e diferenças importantes entre as duas disciplinas:

- 12 Essa notação nos permite determinar a chave da *relação de relacionamento*, conforme discutiremos no Capítulo 7.
- 13 Isso também é verdadeiro para as razões de cardinalidade de relacionamentos binários.
- 14 Todavia, as restrições (min, max) podem determinar as chaves para os relacionamentos binários.
- 15 Uma *ontologia* é alguma coisa similar a um esquema conceitual, mas com mais conhecimento, regras e exceções.

- Ambas as disciplinas usam um processo de abstração para identificar as propriedades comuns e aspectos importantes do minimundo (domínio do discurso), enquanto suprimem as diferenças insignificantes e os detalhes sem importância.
 - Ambas as disciplinas fornecem conceitos, restrições, operações e linguagens para a definição de dados e a representação do conhecimento.
 - O RC possui geralmente âmbito mais abrangente que os modelos de dado semântico. Diferentes formas de conhecimento, como regras (usadas em inferência, dedução e busca), conhecimento incompleto e padrão, e conhecimento temporal e espacial, são representadas em esquemas de RC. Os modelos de banco de dados estão sendo expandidos para incluir alguns desses conceitos (Capítulo 24).
 - Os esquemas de RC incluem os **mecanismos de raciocínio**, que deduzem fatos adicionais de fatos armazenados em um banco de dados. Portanto, enquanto a maioria dos sistemas de banco de dados atuais é limitada a responder a perguntas diretas, os sistemas baseados em conhecimento utilizando esquemas de RC podem responder às consultas que envolvam as **inferências** sobre os dados armazenados. A tecnologia de banco de dados está sendo estendida com mecanismos de inferência (Seção 24.4).
 - Enquanto a maioria dos modelos de dados se concentra na representação de esquemas de banco de dados ou metakonhecimento, os esquemas de RC frequentemente confundem os esquemas com suas próprias instâncias, de maneira a fornecer flexibilidade às exceções de representação. Muitas vezes isso resulta em ineficiências quando esses esquemas de RC são implementados, especialmente quando comparados aos bancos de dados e quando um grande volume de dados (ou fatos) precisa ser armazenado.
- Nesta seção discutiremos quatro **conceitos de abstração** que são usados em ambos os modelos de dados semânticos, como o modelo EER e os esquemas de RC: 1) classificação e instanciação, 2) identificação, 3) especialização e generalização, e 4) agregação e associação. Os conceitos emparelhados, classificação e instanciação, são inversos, como os de generalização e especialização. Os conceitos de agregação e associação também estão relacionados. Discutiremos esses conceitos abstratos e as relações, para suas representações concretas usadas no modelo EER, de modo a esclarecer o processo de abstração de dados e melhorar nossa compreensão dos processos relativos ao projeto conceitual de esquemas. Fecharemos a seção com uma breve discussão sobre o termo *ontologia*, que está sendo largamente utilizado nas recentes pesquisas sobre a representação do conhecimento.

4.8.1 Classificação e Instanciação

O processo de **classificação** envolve, sistematicamente, a correlação de similaridade entre os objetos/entidades com as classes de objeto/tipos entidade. Podemos, assim, descrever (em BD) ou raciocinar (em RC) a respeito de classes em vez de objetos individuais. As coleções de objetos compartilham os mesmos tipos de atributos, relacionamentos e restrições, e pela classificação de objetos simplificamos o processo para a descoberta de suas propriedades. A **instanciação** é o inverso da classificação e se refere à geração e ao exame específico dos diferentes objetos de uma classe. Portanto, uma instância de objeto está relacionada à sua classe de objeto pelo relacionamento **E-UMA-INSTANCIA-DE** ou **E-UM-MEMBRO-DE**. Embora os diagramas da UML não exibam as instâncias, eles admitem uma forma de instanciação, permitindo a exibição de objetos individuais. *Não* descrevemos essa característica em nossa introdução à UML.

Em geral, os objetos de uma classe deveriam ter uma estrutura de tipo similar. Entretanto, alguns objetos podem exibir propriedades que diferem em alguns aspectos de outros objetos da classe; esses **objetos** de exceção também precisam ser modelados, e os esquemas de RC permitem mais variedade de exceções que os modelos de banco de dados. Além disso, certas propriedades se aplicam à classe como um todo, e não a objetos individualmente; os esquemas de RC permitem essas **propriedades de classe**. Os diagramas UML também possibilitam a especificação de propriedades de classe.

No modelo EER, as entidades são classificadas em tipos entidade, de acordo com seus atributos e relacionamentos básicos. Adiante, as entidades são classificadas em subclasses e categorias com base em similaridades e diferenças (exceções) adicionais entre elas. As instâncias de relacionamento são classificadas em tipos relacionamento. Portanto, os tipos entidade, subclasses, categorias e tipos relacionamento são tipos distintos de classes no modelo EER. O modelo EER não proporciona, explicitamente, as propriedades de classes, mas pode ser estendido para fazê-lo. Em UML, os objetos são classificados em classes e isso possibilita exibir tanto as propriedades das classes quanto as dos objetos individualmente.

Os modelos de representação de conhecimento permitem esquemas de classificação múltipla, no qual uma classe é uma *instância* de outra classe (chamada **metaclass**). Observe que isso *não pode* ser representado diretamente no modelo EER porque temos apenas dois níveis — classes e instâncias. O único relacionamento entre as classes no modelo EER é um relacionamento de superclasse/subclasse, enquanto em alguns esquemas de RC um relacionamento adicional classe/instância pode ser representado diretamente em uma hierarquia de classe. Uma instância pode, ela mesma, ser outra classe, possibilitando esquemas de classificação múltiplos níveis.

4.8.2 Identificação

A **identificação** é o processo de abstração pelo qual as classes e os objetos são identificados univocamente por meio de algum **identificador**. Por exemplo, um nome de classe identifica univocamente toda uma classe. Um mecanismo adicional é necessário para diferenciar as instâncias distintas de objetos, por meio do significado dos identificadores de objeto. Além disso, é necessário identificar as diversas manifestações no banco de dados do mesmo objeto do mundo real. Por exemplo, podemos ter uma tupla <Matthew Clarke, 610618,376-9821 > em uma relação PESSOA, e outra tupla <301-540836, CS, 3.8> em uma relação ALUNO, que acontecem para representar a mesma entidade do mundo real. Não há maneira de identificar o fato de que esses dois objetos do banco de dados (tuplas) representam a mesma entidade do mundo real, a menos que façamos o registro, *em tempo de projeto*, da referência cruzada apropriada que forneça essa identificação. Portanto, a identificação é necessária em dois níveis:

- Para a distinção entre os objetos e classes do banco de dados.
- Para identificar os objetos do banco de dados e relacioná-los aos seus correspondentes no mundo real.

No modelo EER, a identificação dos construtores de esquema é baseada em um sistema de nomes únicos para essas construções. Por exemplo, toda classe em um esquema EER — mesmo que seja um tipo entidade, uma subclasse, uma categoria ou um tipo relacionamento — deve ter um nome distinto. Os nomes dos atributos de uma dada classe também devem ser distintos. As regras para evitar a identificação ambígua de nomes de atributos em hierarquias ou reticulados de especializações e generalizações também são necessárias.

Em nível de objeto, os valores dos atributos-chave são usados para distinguir as entidades de um tipo entidade específico. Para os tipos entidade fracas, as entidades são identificadas por uma combinação dos valores de suas chaves parciais próprias e da(s) entidade(s) forte(s) com a(s) qual(is) está(ão) relacionada(s). As instâncias de relacionamento são identificadas por algumas combinações de entidades com as quais se relacionam, dependendo da razão de cardinalidade especificada.

4.8.3 Especialização e Generalização

A especialização é o processo de classificar uma classe de objetos em subclasses mais especializadas. A generalização é o processo inverso de generalizar diversas classes em uma classe abstrata de nível mais alto que inclua os objetos de todas essas classes. A especialização é um refinamento conceitual, enquanto a generalização é uma síntese conceitual. As subclasses são usadas no modelo EER para representar a especialização e a generalização. Chamamos o relacionamento entre uma subclasse e sua superclasse um relacionamento **E-UMA-SUBCLASSE-DE** ou, simplesmente, um relacionamento **E-UM** (IS-A).

4.8.4 Agregação e Associação

A agregação é um conceito de abstração para a construção de objetos compostos a partir de seus objetos componentes. Há três casos em que esse conceito pode ser relacionado ao modelo EER. O primeiro refere-se à situação na qual agregamos os valores do atributo de um objeto para formar o objeto como um todo. O segundo é quando representamos um relacionamento de agregação como um relacionamento comum. O terceiro, que o modelo EER não fornece explicitamente, implica a possibilidade de combinar os objetos que estão relacionados por uma instância de relacionamento em particular em um *objeto agregado de alto nível*. Isso é particularmente vantajoso quando o objeto agregado de alto nível está, ele mesmo, relacionado a outro objeto. Chamamos o relacionamento entre os objetos primitivos e seu objeto agregado **É-UMA-PARTE-DE**; o contrário é chamado **É-UM-COMPONENTE-DE**. A UML fornece todos os três tipos de agregação.

A abstração da associação é usada para associar os objetos de diversas *classes independentes*. Por isso, ela é um tanto similar ao segundo uso da agregação. É representada no modelo EER por tipos relacionamento e, na UML, por associações. Esse relacionamento abstrato é conhecido como **ESTÁ-ASSOCIADO-COM**.

De maneira a entender melhor os diferentes usos da agregação, considere o esquema ER mostrado na Figura 4.14a, que armazena as informações sobre entrevistas para emprego aplicadas a várias empresas. A classe EMPRESA é uma agregação dos atributos (ou objetos componentes) ENome (nome da empresa) e EEndereço (endereço da empresa), ao passo que CANDIDATO_EMPREGO é um agregado de SSN, Nome, Endereço e Telefone. Os atributos de relacionamento NomeContato e Fone-Contato representam o nome e o número do telefone da pessoa, na empresa, responsável pela entrevista. Suponha que algumas entrevistas resultem em ofertas de emprego, enquanto outras, não. Gostaríamos de tratar ENTREVISTA como uma classe para poder associá-la a OFERTA_EMPREGO. O esquema mostrado na Figura 4-14b está *incorreto* porque requer que cada instância do relacionamento entrevista tenha uma oferta de emprego. O esquema mostrado na Figura 4-14c não é permitido, pois o modelo ER não permite relacionamentos entre os relacionamentos (embora a UML o faça).

Uma maneira de representar essa situação é criar uma classe agregada de alto nível, composta por EMPRESA, CANDIDATO_EMPREGO e ENTREVISTA, e relacionar essa classe à OFERTA_EMPREGO, como mostrado na Figura 4-14d. Embora o modelo EER, de acordo com o que foi descrito neste livro, não tenha essa facilidade, alguns modelos de dados semânticos permitem fazer isso e chamam o objeto resultante de **objeto composto** ou **molecular**. Outros modelos tratam dos tipos entidade e tipos relacionamen-to uniformemente, portanto, permitem relacionamentos entre relacionamentos, como ilustrado na Figura 4-Hc.

Para representar essa situação corretamente no modelo ER, conforme foi descrito aqui, precisaríamos criar um novo tipo entidade fraca ENTREVISTA, como pode ser visto na Figura 4.14c, e relacioná-lo a OFERTA_EMPREGO. Assim, poderemos sempre re-presentar essas situações corretamente no modelo ER, por meio da criação de tipos entidade adicionais, embora seja concei-tualmente mais desejável possuir uma representação direta de agregação, como na Figura 4.14d, ou permitir relacionamentos entre relacionamentos, como na Figura 4.14c.

(a)
(b)
(c)
CANDIDATO_EMPREGO
(d)
EMPRESA
CANDIDATCLEMPREGO
OFERTA_EMPREGO

101

FIGURA 4.14 Agregação, (a) O tipo relacionamento ENTREVISTA, (b) Incluindo OFERTA_EMPREGO em um tipo relacionamento ternário (incorreto), (c) O relacionamento RESUUA_EM participando em outros relacionamentos (geralmente não permitido em ER). (d) Usando a agregação e um objeto composto (molecular) (normalmente não permitido em ER). (e) Representação correta em ER.

4.9 Resumo 81

A principal distinção estrutural entre agregação e associação é a seguinte: quando uma instância de associação for deletada, os objetos participantes podem continuar existindo. Entretanto, se suportamos a noção de um objeto agregado — por exemplo, um CARRO, que é construído pelos objetos MOTOR, CHASSI, e PNEUS —, então deletar o objeto agregado CARRO implica deletar todos os seus objetos componentes.

4.8.5 Ontologias e a Web Semântica

Nos últimos anos, o montante de dados computadorizados e as informações disponíveis na Web disparavam de forma descontrolada. Muitos modelos e formatos diferentes são usados. Além dos modelos de banco de dados que apresentamos neste livro, muita informação é armazenada na forma de **documentos**, que são consideravelmente menos estruturados que as informações em um banco de dados. Um projeto de pesquisa que está empenhado em admitir a troca de informações entre os computadores na Web é chamado **Web Semântica**, cuja expectativa é a criação de modelos de representação de conhecimento genéricos o bastante para permitir a troca significativa de conhecimento e busca entre as máquinas. O conceito de *ontologia* é considerado a base mais promissora para atingir os objetivos da Web Semântica e está intimamente ligado à representação do conhecimento. Nesta seção daremos uma breve introdução do que é uma ontologia e como ela pode ser usada como base para a automatização do entendimento, busca e troca de informações.

O estudo de ontologias tenta descrever as estruturas e os relacionamentos que são possíveis na realidade, por meio de algum vocabulário comum e, assim, ser considerado um meio para descrever o conhecimento de uma certa comunidade sobre a realidade. A ontologia é originária da filosofia e da metafísica. Uma definição comumente usada de **ontologia** é "a *especificação* de uma *conceitualização*".

Nessa definição, uma **conceitualização** é o conjunto de conceitos que são utilizados para representar a parte da realidade ou conhecimento que é de interesse de uma comunidade de usuários. A especificação refere-se à linguagem e aos termos de vocabulário que são usados para especificar a conceitualização. A ontologia inclui ambas, a *especificação* e a *conceitualização*. Por exemplo, a mesma conceitualização pode ser especificada em duas linguagens diferentes, resultando duas ontologias separadas. Com base nessa definição bastante geral, não há consenso sobre o que, exatamente, é uma ontologia. Algumas técnicas possíveis para descrever as ontologias que têm sido mencionadas são as seguintes:

- Um **thesaurus** (ou, ainda, um **dicionário** ou um **glossário** de termos) descreve os relacionamentos entre as palavras (vocabulário) que representam vários conceitos.
- Uma **taxonomia** descreve como os conceitos de uma área de conhecimento em particular são relacionados usando-se estruturas similares àquelas utilizadas em uma especialização ou generalização.
- Um **esquema de banco de dados** detalhado é considerado, por alguns, uma ontologia que descreve os conceitos (entidades e atributos) e relacionamentos de um minimundo real.
- Uma **teoria lógica** usa os conceitos de lógica matemática para tentar definir conceitos e seus inter-relacionamentos.

Em geral, os conceitos usados para descrever as ontologias são bastante similares aos que temos discutido na modelagem conceitual, como entidades, atributos, relacionamentos, especializações, e assim por diante. A principal diferença entre uma ontologia e, digamos, um esquema de um banco de dados, é que o esquema está, normalmente, limitado a descrever um subconjunto pequeno de um minimundo real, de maneira a armazenar e gerenciar os dados. Uma ontologia é, geralmente, considerada mais genérica naquilo que ela se empenha em descrever tão completamente quanto possível: uma parcela da realidade.

4.9 RESUMO

Neste capítulo discutimos, primeiro, as extensões para o modelo ER que melhoram sua capacidade de representação. Chamamos o modelo resultante modelo ER estendido ou EER. O conceito de uma subclasse e suas superclasses e o mecanismo relacionado de herança de atributo/relacionamento foram apresentados. Vimos como, às vezes, é necessário criar classes de entidades adicionais, ou em decorrência de atributos específicos adicionais, ou por causa de tipos relacionamento específicos. Discutimos dois processos principais para definir hierarquias e reticulados de superclasse/subclasse: especialização e generalização.

Então mostramos como exibir essas novas construções em um diagrama EER. Também discutimos os vários tipos de restrições que podem ser aplicadas à especialização ou à generalização. As duas principais restrições são total/parcial e disjunção/sobreposição. Além disso, pode ser especificada uma definição por predicado, para uma subclasse ou uma definição por

16 Essa definição é dada em Gruber (1995).

82 Capítulo 4 Modelagem com Entidade-Relacionamento Estendido e UML

atributo para uma especialização. Discutimos as diferenças entre as subclasses definidas pelo usuário e por predicado, e entre as especializações definidas pelo usuário e por atributo. Finalmente discutimos o conceito de uma categoria ou tipo união, que é um subconjunto da união de duas ou mais classes, e demos definições formais de todos esses conceitos apresentados. Introduzimos, então, algumas das notações e terminologias UML para representar a especialização e a generalização. Discutimos também algumas das questões que dizem respeito às diferenças entre os relacionamentos binários e de grau superior, sob quais circunstâncias cada qual deveria ser usado quando projetando um esquema conceitual, e quais tipos diferentes de restrições em relacionamentos n-ário podem ser especificados. Na Seção 4-8 discutimos, brevemente, a disciplina de representação do conhecimento e como ela está relacionada à modelagem de dados semântica. Também fizemos uma avaliação e resumo dos tipos de conceitos de representação de dados abstratos: classificação e instanciação, identificação, especialização e generalização, e agregação e associação. Vimos como os conceitos de EER e UML estão relacionados a cada um deles.

Questões para Revisão

- 4.1. O que é uma subclasse? Quando uma subclasse é necessária na modelagem de dados?
- 4-2. Defina os seguintes termos: *superclasse de uma subclasse*, *relacionamento de superclasse subclasse*, *relacionamento e_um (is-a)*, *especialização*, *generalização*, *categoria*, *atributos específicos (locais)*, *relacionamentos específicos*. 4-3. Discuta o mecanismo de herança de atributo/relacionamento. Por que ele é vantajoso? 4-4. Discuta subclasses definidas pelo usuário e definida por predicado, e identifique as diferenças entre as duas. 4-5. Discuta as especializações definidas pelo usuário e por atributo, e identifique as diferenças entre as duas. 4-6. Discuta os dois tipos principais de restrições em especializações e generalizações.
- 4.7. Qual é a diferença entre uma hierarquia de especialização e um reticulado de especialização?
- 4.8. Qual é a diferença entre especialização e generalização? Por que não exibimos essa diferença nos diagramas de esquemas? 4-9. Como uma categoria difere de uma subclasse regular compartilhada? Para que uma categoria é usada? Ilustre sua resposta com exemplos.
- 4.10. Para cada um dos seguintes termos da UML (seções 3.8 e 4.6), discuta o termo correspondente no modelo EER, se existir algum: *objeto*, *classe*, *associação*, *agregação*, *generalização*, *multiplicidade*, *atributos*, *discriminador*, *ligação*, *atributo de ligação*, *associação reflexiva*, *associação qualificada*.
- 4.11. Discuta as principais diferenças entre as notações para os diagramas de esquema EER e de classe UML, comparando como os conceitos comuns são representados em cada uma.
- 4-12. Discuta as duas notações para a especificação de restrições em relacionamentos n-ário e para que cada uma pode ser usada.
- 4-13. Liste os vários conceitos de abstração de dados e os de modelagem correspondentes no modelo EER.
- 4-14. Qual aspecto de agregação está ausente no modelo EER? Como o modelo EER pode ser estendido, posteriormente, para suportá-lo? 4-15. Quais são as principais similaridades e diferenças entre as técnicas de modelagem conceitual de banco de dados e de representação do conhecimento?
- 4.16. Discuta as similaridades e diferenças entre uma ontologia e um esquema de banco de dados.

Exercícios

- 4.17. Projete um esquema EER para uma aplicação de banco de dados em que você esteja interessado. Especifique todas as restrições que deveriam ser impostas ao banco de dados. Certifique-se de que o esquema tenha, pelo menos, cinco tipos entidade, quatro tipos relacionamento, um tipo entidade fraca, um relacionamento de superclasse/subclasse, uma categoria e um tipo relacionamento n-ário ($n > 2$).
- 4.18. Considere o esquema ER BANCO, da Figura 3.18, e suponha que seja necessário manter o controle de diferentes tipos de CONTAS (CONTA_POUPANCA, CONTA_CORRENTE, ...) e EMPRÉSTIMOS (EMPRESTIMOSJARRO, EMPRESTIMOS_CASA,...). Suponha, ainda, que seja desejável manter o controle das TRANSAÇÕES das contas (depósitos, retiradas, cheques,...) e dos PAGAMENTOS dos empréstimos; ambos incluem quantia, data e hora. Modifique o esquema BANCO usando os conceitos ER e EER de especialização e generalização. Declare todas as considerações que você fizer sobre os requisitos adicionais.
- 4.19 A seguinte narrativa descreve uma versão simplificada da organização das instalações das Olimpíadas, planejadas para os jogos olímpicos de verão. Desenhe um diagrama EER que mostre os tipos entidades os atributos, os relacionamentos e as especializações para essa aplicação. Declare todas as considerações que você fizer. As instalações dos jogos olímpicos estão divididas em complexos de esportes, sendo estes classificados em tipos de esporte *individual* e *multiesporte*. Os complexos de multiesporte têm áreas do complexo designadas para cada esporte, com um indicador de localização

4.9 Resumo 83

(por exemplo, centro, área NE etc). Um complexo tem uma localização, um chefe de organização individual, área total ocupada e outros. Cada complexo envolve uma série de eventos (por exemplo, a pista do estádio pode ser usada para muitas corridas diferentes). Para cada evento há uma data planejada, duração, número de participantes, número de oficiais, e assim por diante. Uma lista de todos os oficiais deverá ser mantida com a lista de eventos em que cada um estará envolvido. E equipamentos diferentes são necessários para os eventos (por exemplo, traves de gol, varas, barras paralelas), bem como para a manutenção. Os dois tipos de instalação (esporte individual e multiesporte) terão tipos diferentes de informação. Para cada tipo, o número de instalações necessárias é mantido com um orçamento aproximado.

4.20 Identifique todos os conceitos importantes representados no estudo de caso do banco de dados da biblioteca descritos aqui. Em particular, identifique as abstrações de classificação (tipos entidade e tipos relacionamento), agregação, identificação e especialização/generalização. Sempre que possível, especifique as restrições de cardinalidade (min, max). Liste os detalhes que eventualmente afetarão o projeto, mas que não se referem ao projeto conceitual. Liste também as restrições semânticas separadamente. Desenhe o diagrama EER do banco de dados da biblioteca. **Estudo de Caso:** A Biblioteca Geórgia Tech — Geórgia Tech Library (GLT) — tem aproximadamente 16 mil sócios, 100 mil títulos e 250 mil volumes (ou uma média de 2,5 cópias por livro). Cerca de 10% dos volumes estão sempre emprestados. Os bibliotecários garantem que os livros que os sócios quiserem pegar emprestado estarão disponíveis quando desejarem. Assim, os bibliotecários devem saber quantas cópias de cada livro encontram-se na biblioteca, ou estão emprestadas, em um dado momento. Um catálogo dos livros está disponível on-line, constando os livros por autor, título e assunto. Para cada título da biblioteca há uma descrição no catálogo que pode ter de uma frase a diversas páginas. Os bibliotecários de referência querem ter acesso a essa descrição quando os sócios pedem informações sobre um livro. O pessoal da biblioteca divide-se em bibliotecário-chefe, bibliotecários associados aos departamentos, bibliotecários de referência, pessoal de verificação e assistentes de bibliotecário.

Os livros podem ser emprestados por 21 dias. Aos sócios é permitida a retirada de, no máximo, cinco livros por vez. Geralmente, eles devolvem os livros em três ou quatro semanas. A maioria deles sabe que tem uma semana de tolerância antes que uma notificação lhes seja enviada, por isso tentam devolver o livro emprestado antes do término desse período. Cerca de 5% dos sócios precisam ser lembrados sobre a devolução de um livro. A maior parte dos livros em atraso é devolvida dentro de um mês, a partir da data da dívida. Aproximadamente, 5% dos livros em atraso são perdidos ou nunca devolvidos. A maioria dos sócios ativos da biblioteca é composta por aqueles que fazem empréstimo, pelo menos, dez vezes durante o ano. Um por cento dos sócios faz 15% dos empréstimos, e 10% dos sócios fazem 40% dos empréstimos. Cerca de 20% dos sócios são totalmente inativos; e apesar de serem sócios, nunca fazem empréstimos. Para se tornar um sócio da biblioteca, os requerentes preenchem um formulário informando seu SSN, endereço no *compus* e residencial para correspondência e números de telefone. Os bibliotecários emitem, então, um cartão eletrônico com a foto do sócio. Esse cartão é válido por quatro anos. Um mês antes de o cartão expirar, uma notificação para a renovação é enviada ao sócio. Os professores do instituto são, automaticamente, considerados sócios. Quando um novo docente ingressa no instituto, suas informações são preenchidas nos registros de empregados e um cartão da biblioteca é enviado ao seu endereço no *compus*. Aos professores é permitida a retirada de livros por um intervalo de três meses, com um período de tolerância de duas semanas. As notificações de renovação para os professores são enviadas ao endereço no *compus*.

A biblioteca não empresta determinados livros, como os de referência, livros raros e mapas. Os bibliotecários devem distinguir entre os livros que podem ser emprestados, e aqueles que não podem ser emprestados. Além disso, mantêm uma lista com alguns livros que desejariam adquirir, mas que não conseguem obter, como livros raros ou esgotados e livros que foram perdidos ou destruídos e que não foram substituídos. Os bibliotecários devem ter um sistema de controle para os livros que não podem ser emprestados, bem como para aqueles que têm interesse em adquirir. Alguns livros podem ter o mesmo título, portanto, o título não pode ser usado como identificação. Todo livro é identificado por seu Número de Livro Padrão Internacional — International Standard Book Number (ISBN) —, um código internacional único, designado a todos os livros. Dois livros com o mesmo título podem ter diferentes ISBNs, se estiverem em idiomas diferentes, ou se tiverem diferentes acabamentos (capa dura ou brochura). As edições do mesmo livro têm ISBNs diferentes. O sistema de um banco de dados proposto deve controlar os sócios, os livros, o catálogo e as atividades de empréstimo.

4.21. Projete um banco de dados para controlar as informações de um museu de arte. Suponha que os seguintes requisitos sejam coletados:

- O museu tem uma coleção de OBJETO_ARTE. Cada OBJETO_ARTE tem um único NumId, um Artista (se conhecido), um Ano (quando foi criado, se conhecido), um Título e uma Descrição. Os objetos de arte são categorizados de diversas formas, conforme apresentado a seguir.

84 Capítulo 4 Modelagem com Entidade-Relacionamento Estendido e UML

- Os OBJETOS_ARTE são categorizados de acordo com o tipo. Há três principais: PINTURA, ESCULTURA e ESTATUARIA, e outro tipo chamado OUTROS, para acomodar os objetos que não se enquadram em um dos três tipos principais.
- Uma PINTURA tem um TipoTinta (óleo, aquarela etc), o suporte em que é Desenhada (papel, tela, madeira etc.) e Estilo (moderno, abstrato etc).
- Uma ESCULTURA ou uma ESTATUARIA tem um Material do qual foi criada (madeira, pedra etc), Altura, Peso e Estilo.
- Um objeto de arte da categoria OUTROS possui um Tipo (gravura, foto etc.) e Estilo.
- Os OBJETOS_ARTE também são categorizados como COLECAO_PERMANENTE, que são propriedade do museu (contém informação sobre a DataAquisicao, se estão EmExposicao ou no depósito e o Custo), ou EMPRESTADO, que possuiu informação referente à Coleção (da qual foi emprestado), DataEmprestimo e DataDevolucao.
- Os OBJETOS_ARTE também têm informação do país/cultura, usando a informação do país/cultura de Origem (italiana, egípcia, norte-americana, indiana etc.) e Período (Renascimento, Modernismo, Antigüidade etc).
- O museu controla as informações a respeito do ARTISTA, se conhecido: Nome, DataNasc (se conhecida), DataMorte, PaisdeOrigem, Período, EstiloPrincipal e Descrição. Admite-se o Nome como único.
- Diferentes EXPOSIÇÕES ocorrem, cada uma tendo um Nome, DataInicio e DataFinal. As EXPOSIÇÕES estão relacionadas a todos os objetos de arte que estiveram em exibição durante o evento.
- São mantidas as informações de outras COLEÇÕES com as quais o museu mantém contato, incluindo Nome (único), Tipo (museu, pessoal etc), Descrição, Endereço, Telefone e atual PessoaContato.

Desenhe um diagrama do esquema EER para essa aplicação. Discuta todas as considerações feitas por você e justifique suas escolhas para o projeto EER.

4-22. A Figura 4.15 mostra um diagrama EER exemplo para um banco de dados de um pequeno aeroporto privado, que é usado para o controle dos aviões, seus proprietários, empregados e pilotos. Dos requisitos para esse banco de dados foram coletadas as seguintes informações: cada AVIÃO tem um número de registro [Reg#], é de um tipo em particular [DEJTPO] e está armazenado em um hangar [ARMAZENADO_EM]. Cada TIPO_AVIAO possui um número de modelo [Modelo], uma capacidade [Capacidade] e um peso [Peso]. Cada HANGAR tem um número [Número], capacidade [Capacidade] e localização [Localização]. O banco de dados também controla os PROPRIETÁRIOS de cada avião [POSSUI] e os EMPREGADOS que mantêm o avião [MANTÉM]. Cada instância de relacionamento em POSSUI vincula um avião ao proprietário e inclui a data da compra [da-taC]. Cada instância de relacionamento em MANTÉM liga um empregado a um registro de serviço [SERVIÇO]. Cada avião passa por serviços muitas vezes, portanto, está relacionado por [SERVICOJWIAO] a um número de registros de serviço. Um registro de serviço inclui como atributos a data da manutenção [Data], o número de horas gastas no trabalho [Horas], e o tipo de trabalho feito [Códigotrabalho]. Usamos um tipo entidade fraca [SERVIÇO], para representar o serviço do avião, pois o número do registro do avião é usado para identificar um registro de serviço. Um proprietário é uma pessoa ou uma corporação. Por isso usamos um tipo união (categoria) [PROPRIETÁRIO], que é um subconjunto da união dos tipos entidade corporação [CORPORAÇÃO] e pessoa [PESSOA]. Os pilotos [PILOTO] e os empregados [EMPREGADO] são subclasses de PESSOA. Cada piloto tem atributos específicos, como número da licença [Num_Lic] e restrições [Restr]; cada empregado possui atributos específicos, como salário [Salário] e turno trabalhado [Turno]. Todas as entidades PESSOA no banco de dados têm dados mantidos em número do seguro social [SSN], nome [Nome], endereço [Endereço] e número de telefone [Fone]. Para as entidades CORPORAÇÃO, os dados mantidos incluem nome [Nome], endereço [Endereço] e número de telefone [Fone]. O banco de dados também mantém o controle dos tipos de aviões com os quais cada piloto está autorizado a voar [VOA], e os tipos de aviões nos quais cada empregado pode trabalhar na manutenção [TRABALHA_EM]. Mostre como o esquema EER do PEQUENO AEROPORTO da Figura 4-15 pode ser representado na notação UML. (Nota: Não discutimos como representar as categorias tipos união em UML, assim, você não precisa mapeá-las nesta e na questão seguinte).

4-23. Mostre como o esquema EER da UNIVERSIDADE, da Figura 4.9, pode ser representado na notação UML.

Bibliografia Seleccionada

Muitos artigos têm proposto modelos de dados conceituais ou semânticos. Fornecemos aqui uma lista representativa. Uma compilação de artigos, incluindo os de Abrial (1974), modelo DIAM de Senko (1975), método NIAM (Verheijen e Van-Bekkm, 1982) e Bracchi et al. (1976) apresenta os modelos semânticos baseados no conceito de relacionamentos binários. Outra coleção mais antiga de artigos discute os métodos para estender o modelo relacional, a fim de aumentar suas capacidades de modelagem. Isso inclui os artigos de Schmid e Swenson (1975), Navathe e Schkolnick (1978), o modelo RM/T de Codd (1979), Furtado (1978) e o modelo estrutural de Wiederhold e Elmasri (1979).

O modelo ER foi proposto, originalmente, por Chen (1976) e é formalizado em Ng (1981). Desde essa época, numerosas extensões para sua capacidade de modelagem têm sido propostas, como em Scheuermann et al. (1979), Dos Santos et al.

4.9 Resumo 85

(1979), Teorey *et al.* (1986), Gogolla e Hohenstein (1991) e o modelo de relacionamento entidade-categoria (ECR) de Elmasri *et al.* (1985). Smithe Smith (1977) apresentaram conceitos de generalização e agregação. O modelo de dados semânticos de Hammer e McLeod (1981) introduziu os conceitos de reticulados de classe/subclasse, bem como outros conceitos de modelagem avançada.

FIGURA 4.15 Esquema EER para um banco de dados de um PEQUENO AEROPORTO.

Uma pesquisa de modelagem de dados semânticos aparece em Hull e King (1987). Eick (1991) discute projeto e transformações dos esquemas conceituais. A análise de restrições para os relacionamentos n-ário é dada em Soutou (1998). A UML é descrita em detalhes em Booch, Rumbaugh e Jacobson (1999). Fowler e Scott (2000) e Stevens e Pooley (2000) dão introduções concisas aos conceitos UML.

Fensel (2000) é uma boa referência para Web Semântica. Uschold e Gruninger (1996) e Gruber (1995) discutem ontologias. Uma recente publicação completa da *Communications* da ACM é dedicada aos conceitos e aplicações de ontologia.

MODELO RELACIONAL: CONCEITOS, RESTRICÇÕES, LINGUAGENS, DESIGN E PROGRAMAÇÃO

5

O Modelo de Dados Relacional e as Restrições de um Banco de Dados Relacional

Este capítulo inicia a Parte II do livro, com uma abordagem sobre os bancos de dados relacionais. O modelo relacional foi introduzido por Ted Codd, da IBM Research, em 1970, em um artigo clássico (Codd, 1970) que imediatamente atraiu a atenção em virtude de sua simplicidade e base matemática. O modelo usa o conceito de uma *relação matemática* — algo como uma tabela de valores — como seu bloco de construção básica e tem sua base teórica na teoria dos conjuntos e na lógica de predicados de primeira ordem. Neste capítulo discutiremos as características básicas do modelo e suas restrições.

As primeiras implementações comerciais do modelo relacional tornaram-se disponíveis no início da década de 80, com o o SGBD Oracle e o sistema SQL/DS do sistema operacional MVS, da IBM. Desde essa época, o modelo tem sido implementado em um grande número de sistemas comerciais. Os SGBDs relacionais (SGBDRs) mais conhecidos atualmente são o DB2 e o Informix Dynamic Server (da IBM), o Oracle e o Rdb (da Oracle), e o SQL Server e o Access (da Microsoft).

Em razão da importância do modelo relacional, dedicamos toda a Parte II deste livro a esse modelo e às linguagens associadas a ele. O Capítulo 6 cobre as operações de álgebra relacional e introduz a notação de cálculo relacional para dois tipos de cálculo — o de tupla e o de domínio. O Capítulo 7 relaciona as estruturas de dados do modelo relacional aos construtores do modelo ER ou EER, e apresenta os algoritmos para planejamento de um esquema de banco de dados relacional, mapeando um esquema conceitual no modelo ER ou EER (capítulos 3 e 4) em uma representação relacional. Esses mapeamentos são incorporados em muitos projetos de banco de dados e ferramentas CASE. No Capítulo 8 descrevemos a linguagem de consulta SQL, que é o *padrão* para os SGBDs relacionais comerciais. O Capítulo 9 discute as técnicas de programação usadas para acessar os sistemas de banco de dados e apresenta tópicos adicionais relacionados à linguagem SQL — restrições, visões e a noção de conexão a um banco de dados relacional via protocolos-padrão ODBC e JDBC. Os capítulos 10 e 11 da Parte III do livro apresentam outro aspecto do modelo relacional, chamado restrições formais de dependências funcionais ou multivaloradas. Essas dependências são usadas para desenvolver uma teoria de projeto de um banco de dados relacional baseada no conceito conhecido como *normalização*.

Os modelos de dados que precederam o modelo relacional compreendem os modelos hierárquico e de rede. Foram propostos na década de 60 e implementados em antigos SGBDs durante as décadas de 70 e 80. Em virtude de sua importância histórica e da grande base de usuários existente para esses SGBDs, incluímos um resumo dos pontos de especial interesse desses modelos em apêndices, que estão disponíveis no site do livro. Esses modelos e sistemas estarão conosco por muitos anos e serão, a partir de agora, referidos como *sistemas de banco de dados legados*.

Neste capítulo, concentraremos-nos na descrição dos princípios básicos do modelo relacional de dados. Começaremos definindo os conceitos de modelagem e a notação do modelo relacional, na Seção 5.1. A Seção 5.2 é dedicada à discussão de restrições relacionais, que são, agora, consideradas uma importante parte do modelo relacional, sendo automaticamente garantidas

1 CASE significa engenharia de software auxiliada por computador (*computer-aided software engineering*).

pela maioria dos SGBDs relacionais. A Seção 5.3 define as operações de atualização do modelo relacional e discute como as violações de restrições de integridade são tratadas.

5.1 CONCEITOS DO MODELO RELACIONAL

O modelo relacional representa o banco de dados como uma coleção de *relações*. Informalmente, cada relação se parece com uma tabela de valores ou, em alguma extensão, com um arquivo de registros 'plano'. Por exemplo, o banco de dados dos arquivos que foram mostrados na Figura 1.2 é similar à representação do modelo relacional. Entretanto, há importantes diferenças entre relações e arquivos, conforme veremos em breve.

Quando uma relação é pensada como uma **tabela** de valores, cada linha na tabela representa uma coleção de valores de dados relacionados. Introduzimos, no Capítulo 3, os tipos entidade e os tipos relacionamento como conceitos para modelagem de dados do mundo real. No modelo relacional, cada linha na tabela representa um fato que corresponde a uma entidade ou relacionamento do mundo real. O nome da tabela e os nomes das colunas são usados para ajudar na interpretação do significado dos valores em cada linha. Por exemplo, a primeira tabela da Figura 1.2 é chamada ALUNO, pois cada linha representa os fatos sobre uma entidade aluno em particular. Os nomes das colunas — Nome, NumeroAluno, Classe e Curso_Hab — especificam como interpretar os valores de dados em cada linha, com base na coluna em que cada valor está. Todos os valores em uma coluna são do mesmo tipo de dado.

Na terminologia do modelo relacional formal, uma linha é chamada *tupla*, um cabeçalho de coluna é conhecido como *atributo*, e a tabela é chamada *relação*. O tipo de dado que descreve os tipos de valores que podem aparecer em cada coluna é representado pelo *domínio* de valores possíveis. Definimos, agora, esses termos — *domínio*, *tupla*, *atributo* e *relação* — mais precisamente.

5.1.1 Domínios, Atributos, Tuplas e Relações

Um **domínio** D é um conjunto de valores atômicos. Por **atômico** entendemos que cada valor no domínio é indivisível no que diz respeito ao modelo relacional. Um método comum para a especificação de um domínio é definir um tipo de dado do qual os valores de dados que formam o domínio sejam retirados. Também é útil especificar um nome para esse domínio, de modo a ajudar na interpretação de seus valores. Seguem alguns exemplos de domínios:

- Numeros_fone_EUA: o conjunto de números de telefone válido nos Estados Unidos, com dez dígitos.
- Numeros_fone_local: o conjunto de números de telefone de sete dígitos, válido para um código de área em particular nos Estados Unidos.
- Numeros_seguro_social: o conjunto de nove dígitos válidos do número do seguro social.
- Nomes: o conjunto de cadeias de caracteres que representa os nomes de pessoas.
- Medias_pontos_graduacao: possíveis valores de médias computadas de pontos para a graduação; cada um deve ser um número real (ponto-flutuante) entre zero e quatro.
- Idades_empregado: possíveis idades dos empregados de uma empresa; cada um deve ter um valor entre 15 e 80 anos de idade.
- Nomes_departamento_academico: o conjunto dos nomes dos departamentos acadêmicos em uma universidade, como ciência da computação, economia e física.
- Codigos_departamento_academico: o conjunto de códigos dos departamentos acadêmicos, como CC, ECON e FIS.

Os precedentes são chamados definições *lógicas* de domínios. Um **tipo de dado** ou **formato** também é especificado para cada domínio. Por exemplo, o tipo de dado para o domínio Numeros_fone_EUA pode ser declarado como uma cadeia de caracteres no formato *(ddd)ddd-dddd*, em que cada *d* é um dígito numérico (decimal), e os três primeiros dígitos formam um código de área de telefone válido. O tipo de dado para Idades_empregado é um número inteiro entre 15 e 80. Para Nomes_departamento_academico, o tipo de dado é o conjunto de todas as cadeias de caracteres que representam nomes válidos de departamentos. Assim, a um domínio é dado um nome, tipo de dado e formato. Informações adicionais para interpretação dos valores de um domínio também podem ser fornecidas; por exemplo, um domínio numérico como Pesos_pessoa deveria ter uma unidade de medida, como libras ou quilogramas.

Um **esquema de relação** R, indicada por $R(A_1, A_2, \dots, A_n)$, é composto de um nome de relação R e de uma lista de atributos A_1, A_2, \dots, A_n . Cada **atributo** A é o nome de um papel desempenhado por algum domínio D no esquema de relação R.

5.1 Conceitos do Modelo Relacional 91

D é chamado **domínio** de A_i , e é indicado por $\text{dom}(A_i)$. Um esquema de relação é usado para *descrever* uma relação; R é chamado **nome** de sua relação. O **grau** (ou **arity**) de uma relação é o número n de atributos de seu esquema de relação.

Um exemplo de um esquema de relação para uma relação de grau sete, que descreve os universitários, é o seguinte:

ALUNO (Nome, SSN, FoneResidencia, Endereço, FoneEscritorio, Idade, MPG)

Usando o tipo de dado de cada atributo, a definição, algumas vezes, é escrita como:

ALUNO (Nome: *string*, SSN: *string*, FoneResidencia: *string*, Endereço: *string*, FoneEscritorio: *string*, Idade: *integer*, MPG: *real*)

Para esse esquema de relação, ALUNO é o nome da relação, que tem sete atributos. Na definição anterior, mostramos a designação de tipos genéricos, como *string* ou *integer* para os atributos. Mais precisamente, podemos especificar os seguintes domínios definidos previamente para alguns dos atributos da relação ALUNO: $\text{dom}(\text{Nome}) = \text{Nomes}$; $\text{dom}(\text{SSN}) = \text{Numeros_seguro_social}$; $\text{dom}(\text{FoneResidencia}) = \text{Numeros_fone_local}$, $\text{dom}(\text{FoneEscritorio}) = \text{Numeros_fone_local}$, e $\text{dom}(\text{MPG}) = \text{Medias_pontos_graduacao}$. Também é possível referir-se a atributos de um esquema de relação por sua posição na relação, assim, o segundo atributo da relação ALUNO é SSN, enquanto o quarto atributo é Endereço.

Uma **relação** (ou **estado da relação**) r do esquema de relação $R(A_1, A_2, \dots, A_n)$, indicado por $r(R)$, é um conjunto de n -tuplas $r = \{t_1, t_2, \dots, t_m\}$. Cada **n -tupla** têm uma lista ordenada de n valores $t = \langle v_1, v_2, \dots, v_n \rangle$, em que cada valor v_i , $1 < i < n$, é um elemento do $\text{dom}(A_i)$ ou um valor **null** especial. O $j^{\text{ésimo}}$ valor na tupla t , que corresponde ao atributo A_i , é referido como $t[A_j]$ (ou $t[i]$, se usarmos a notação posicional). Os termos **intenção de relação** para o esquema R e **extensão de relação** para um estado de relação $r(R)$ também são comumente usados.

A Figura 5.1 mostra um exemplo de uma relação ALUNO que corresponde ao esquema ALUNO já especificado. Cada tupla na relação representa uma entidade aluno em particular. Exibimos a relação como uma tabela, na qual cada tupla é mostrada como uma *linha* e cada atributo corresponde a um *cabeçalho de coluna*, indicando um papel ou interpretação de valores dessa coluna. Os *valores null* representam os atributos cujos valores são desconhecidos ou não existem para alguma tupla individual de ALUNO.

Nome da relação

Atributos

ALUNO	Nome	SSN	FoneResidencia	Endereço	FoneEscritorio	Idade	MPG
Tuplas: $r(R) \rightarrow$	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25

FIGURA 5.1 Os atributos e as tuplas de uma relação ALUNO.

A definição anterior de uma relação pode ser *redefinida* mais formalmente como segue. Uma relação (ou estado da relação) $r(R)$ é uma **relação matemática** de grau n nos domínios $\text{dom}(A_1), \text{dom}(A_2), \dots, \text{dom}(A_n)$, que é um **subconjunto** do **produto cartesiano** dos domínios que definem R :

$r(R) \subseteq (\text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n))$

O produto cartesiano especifica todas as possíveis combinações de valores dos domínios subjacentes. Então, se indicamos o número total de valores, ou **cardinalidade**, em um domínio D por $|D|$ (presumindo que todos os domínios sejam finitos), o número total de tuplas no produto cartesiano é

$|\text{dom}(A_1)| \times |\text{dom}(A_2)| \times \dots \times |\text{dom}(A_n)|$

De todas essas possíveis combinações, um estado de relação em um dado momento — **estado de relação corrente** — reflete apenas as tuplas válidas que representam um estado em particular do mundo real. Em geral, como o estado do mundo

3 Com o grande aumento dos números de telefones, causado pela proliferação dos telefones móveis, algumas áreas metropolitanas têm, agora, diversos códigos de área, tanto que a discagem local de sete dígitos tem sido suspensa. Nesse caso, poderíamos usar como domínio os *Numeros_fone_EUA*.

4 Também chamada de **instância** de relação. Não usaremos esse termo porque a palavra *instância* será usada para referir-se a uma tupla ou linha única.

92 Capítulo 5 O Modelo de Dados Relacional e as Restrições de um Banco de Dados Relacional

real muda, assim também a relação se transforma em outro estado de relação. Entretanto, o esquema R é relativamente estático e não muda, exceto muito raramente — por exemplo, por meio da adição de um atributo para representar uma nova informação que, originalmente, não era armazenada na relação.

É possível que muitos atributos possuam *o mesmo domínio*. Os atributos indicam diferentes papéis, ou interpretações, para um domínio. Por exemplo, na relação *ALUNO*, o mesmo domínio *Numero_fone_local* desempenha o papel de *FoneResidencia*, referindo-se ao 'telefone residencial do aluno', e o papel de *FoneEscritorio*, ao 'fone do escritório do aluno.'

5.1.2 Características das Relações

A definição anterior de relações implica certas características que fazem uma relação diferente de um arquivo ou de uma tabela. Discutiremos, agora, algumas dessas características.

Ordenação de Tuplas em uma Relação. Uma relação é definida como um *conjunto* de tuplas. Matematicamente, os elementos de um subconjunto *não* têm *ordem* entre eles, portanto, as tuplas em uma relação não têm qualquer ordem em particular. Entretanto, em um arquivo, os registros são fisicamente armazenados em disco (ou na memória), então, sempre há uma ordem entre os registros. Essa ordenação indica o primeiro, o segundo, *i*-ésimos e o último registro no arquivo. Analogamente, quando exibimos uma relação em uma tabela, as linhas são exibidas em uma certa ordem.

A ordenação de tupla não é parte da definição de uma relação, porque uma relação se esforça para representar fatos em um nível lógico ou abstrato. Muitas ordens lógicas podem ser especificadas em uma relação. Por exemplo, as tuplas da relação *ALUNO*, na Figura 5.1, poderiam ser ordenadas logicamente pelos valores de Nome, ou SSN, ou Idade, ou por algum outro atributo. A definição de uma relação não especifica qualquer ordem: *não* há *preferência* por uma ordenação lógica sobre outra. Por isso a relação exibida na Figura 5.2 é considerada *idêntica* àquela mostrada na Figura 5.1. Quando uma relação é implementada como um arquivo ou exibida como uma tabela, uma ordenação em particular pode ser especificada nos registros do arquivo ou linhas da tabela.

Ordenação de Valores Dentro de uma Tupla e uma Definição Alternativa de uma Relação. De acordo com a definição anterior de uma relação, uma *n*-tupla é uma lista *ordenada* de *n* valores, tanto assim que a ordenação de valores em uma tupla — ou seja, dos atributos no esquema da relação — é importante. Entretanto, em nível lógico, a ordem dos atributos e seus valores *não* são tão importantes, enquanto a correspondência entre atributos e valores for mantida.

Uma definição alternativa de uma relação pode ser dada, tornando *desnecessária* a ordenação de valores em uma tupla. Nessa definição, um esquema de relação $R = \{A_1, A_2, \dots, A_n\}$ é um conjunto de atributos, e um estado da relação $r(R)$, um conjunto finito de mapeamentos $r = \{t_1, t_2, \dots, t_m\}$, em que cada tupla t_i é um mapeamento de R para D , e D , a união dos domínios do atributo, isto é, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \dots \cup \text{dom}(A_n)$. Nessa definição, $t[A_i]$ deve estar em $\text{dom}(A_i)$ para $1 \leq i \leq n$ para cada mapeamento t em r . Cada mapeamento t_i é conhecido como uma tupla.

De acordo com essa definição de tupla como um mapeamento, uma tupla pode ser considerada um conjunto de pares (*<atributo>*, *<valor>*), em que cada par fornece o valor do mapeamento de um atributo A_i para um valor v_i do $\text{dom}(A_i)$. A ordenação dos atributos *não* é importante, porque o nome do atributo aparece com seu valor. Por definição, as duas tuplas mostradas na Figura 5.3 são idênticas. Isso faz sentido em nível abstrato ou lógico, uma vez que não há razão para preferir que, em uma tupla, o valor de um atributo apareça antes do outro.

Quando uma relação é implementada como um arquivo, os atributos são fisicamente ordenados como campos dentro de um registro. Geralmente usamos a primeira definição de relação, em que os atributos e os valores nas tuplas *estão ordenados*, pois isso torna a notação mais simples. Entretanto, a definição alternativa dada aqui é mais geral.

Valores e Nulls nas Tuplas. Cada valor em uma tupla é um valor atômico, isto é, não é divisível em componentes dentro da estrutura do modelo relacional básico. Por isso os atributos compostos e multivalorados (Capítulo 3) não são permitidos. Esse modelo é chamado, às vezes, modelo relacional plano. Muito da teoria por trás do modelo relacional foi desenvolvida com essa conjectura em mente, chamada pressuposto da primeira forma normal. Por isso os atributos multivalorados devem ser representados em relações separadas, e os atributos compostos são representados, no modelo relacional básico, apenas por seus atributos componentes simples.

5 Como veremos, a definição alternativa de relação será útil quando discutirmos o processamento de consulta nos capítulos 15 e 16.

6 Discutiremos essa pressuposição com mais detalhes no Capítulo 10.

7 As extensões do modelo relacional removem essas restrições. Por exemplo, os sistemas objeto-relacionais permitem os atributos complexo-estruturados como não-primeira forma normal ou modelos relacionais aninhados, como veremos no Capítulo 22.

5.1 Conceitos do Modelo Relacional 93

ALUNO	Nome	SSN	FoneResidencia	Endereço	FoneEscritorio	Idade	MPG
	Dick Davidson	422-11-2320	<i>null</i>	3452 Elgin Road	749-1253	25	3.53
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	<i>null</i>	19	3.25
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	<i>null</i>	18	2.89
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	<i>null</i>	19	3.21

FIGURA 5.2 A relação ALUNO da Figura 5.1 com uma ordenação diferente das tuplas.

$t = \langle (\text{Nome}, \text{Dick Davidson}), (\text{SSN}, 422-11-2320), (\text{FoneResidencia}, \text{null}), (\text{Endereço}, 3452 \text{ Elgin Road}),$

$(\text{FoneEscritorio}, 749-1253), (\text{Idade}, 25), (\text{MPG}, 3.53) \rangle$

$t = \langle (\text{Endereço}, 3452 \text{ Elgin Road}), (\text{Nome}, \text{Dick Davidson}), (\text{SSN}, 422-11-2320), (\text{Idade}, 25), (\text{FoneEscritorio}, 749-1253), (\text{MPG}, 3.53), (\text{FoneResidencia}, \text{null}) \rangle$

FIGURA 5.3 Duas tuplas idênticas quando a ordenação dos atributos e dos valores não faz parte da definição de relação.

Um importante conceito se refere a **nulls**, que são usados para representar os valores de atributos que podem ser desconhecidos ou não se aplicar a uma tupla. Um valor especial, chamado **null**, é usado para esses casos. Por exemplo, na Figura 5.1, algumas tuplas dos alunos têm **null** para seus fones de escritório, porque eles não têm um escritório (isto é, o fone do escritório *não se aplica* a esses alunos). Outros alunos dispõem de **null** para o fone residencial, presumivelmente porque eles não possuem um fone residencial ou têm um, mas não o conhecemos (o valor é *desconhecido*). Em geral, podemos ter *diversos significados* para os valores **null**, como 'valor desconhecido', 'valor existe, mas não está disponível' ou 'atributo não se aplica a essa tupla'. Um exemplo desse último tipo de **null** ocorrerá se adicionarmos um atributo *Visto_status* à relação ALUNO, que se aplica apenas às tuplas que representam os alunos estrangeiros. É possível planejar diferentes códigos para diferentes significados de valores **null**. A incorporação de diferentes tipos de valores **null** nas operações de modelo relacional (Capítulo 6) tem apresentado dificuldades e está fora do escopo da nossa discussão.

Interpretação (Sentido) de uma Relação. O esquema de relação pode ser interpretado como uma declaração ou um tipo de asserção. Por exemplo, o esquema da relação ALUNO da Figura 5.1 declara que, em geral, uma entidade aluno tem um Nome, SSN, FoneResidencia, Endereço, FoneEscritorio, Idade e MPG. Cada tupla na relação pode, então, ser interpretada como um **fato** ou uma instância em particular da asserção. Por exemplo, a primeira tupla da Figura 5.1 estabelece o fato de que há um aluno cujo nome é Benjamin Bayer, seu SSN é 305-61-2435, sua Idade é 19, e assim por diante.

Observe que algumas relações devem representar fatos sobre *entidades*, ao passo que outras relações devem representar fatos sobre *relacionamentos*. Por exemplo, o esquema de relação CURSO/HAB (SSNALuno, CódigoDepartamento) estabelece que os alunos fazem seus cursos em departamentos acadêmicos. Uma tupla nessa relação liga um aluno ao departamento de seu curso. Por isso o modelo relacional representa os fatos sobre ambos, entidades e relacionamentos, *indistintamente*, como relações. Às vezes, esse fato compromete o entendimento, pois é preciso supor se uma relação representa um tipo entidade ou um tipo relacionamento. Os procedimentos de mapeamento no Capítulo 7 mostram como diferenciar as construções de modelos ER e EER, convertidos em relações.

Uma interpretação alternativa para um esquema de relação é entendê-la como um predicado — nesse caso, os valores em cada tupla são interpretados como valores que *satisfazem* o predicado. Essa interpretação é bastante útil no contexto de linguagens lógicas de programação, como Prolog, pois permite que o modelo relacional seja usado com essas linguagens (Seção 24.4).

5.1.3 Notação do Modelo Relacional

Usaremos a seguinte notação em nossa apresentação:

- Um esquema de relação R de grau n é indicado por $R(A_1, A_2, \dots, A_n)$.
- Uma n -tupla tem uma relação $r(R)$ é indicada por $t = \langle v_1, v_2, \dots, v_n \rangle$, em que v_i é o valor correspondente ao atributo A_i . A seguinte notação refere-se aos **valores componentes** das tuplas:
 - Ambos, $t[A_i]$ e $t.A_i$ (e, algumas vezes, $t[i]$), referem-se ao valor v_i em t do atributo A_i .
 - Ambos, $t[A_1, A_2, \dots, A_j]$ e $t(A_1, A_2, \dots, A_j)$, em que A_1, A_2, \dots, A_j é uma lista de atributos de R , fazem referência à sub tupla de valores $\langle v_1, v_2, \dots, v_j \rangle$ de t , correspondentes aos atributos especificados na lista.
- A_s letras Q , R , S significam nomes de relação.

- As letras q, r, s significam estados da relação.
 - As letras t, u, v significam tuplas.
 - Em geral, o nome de um esquema de relação, como ALUNO, *também indica o conjunto corrente de tuplas nessa relação — o estado corrente da relação* —, enquanto ALUNO(Nome, SSN, ...) refere-se *apenas* ao esquema da relação.
 - Um atributo A pode ser qualificado com o nome de relação R à qual ele pertence pelo uso da notação *de ponto* R.A — por exemplo, ALUNO.Nome ou ALUNO.Idade. Isso porque o mesmo nome pode ser usado por dois atributos em diferentes relações. Entretanto, todos os nomes de atributos *em uma relação em particular* devem ser distintos.
- Como um exemplo, considere a tupla $t = \langle \text{'Barbara Benson'}, '533-69-1238', '839-8461', 7384 \text{ Fontana Lane'}, \text{nuii}, 19,3.25 \rangle$ da relação ALUNO da Figura 5.1. Temos $t[\text{Nome}] = \langle \text{'Barbara Benson'} \rangle$, e $t[\text{SSN, MPG, Idade}] = \langle '533-69-1238', 3.25, 19 \rangle$.

5.2 RESTRIÇÕES DO MODELO RELACIONAL E ESQUEMAS DE UM BANCO DE DADOS RELACIONAL

Até aqui discutimos as características de uma única relação. Em um banco de dados relacional, normalmente existirão muitas relações, e as tuplas dessas relações estão, em geral, relacionadas de várias maneiras. O estado do banco de dados como um todo corresponderá aos estados de todas as suas relações em um determinado instante. Há, geralmente, muitas limitações ou restrições para os valores reais em um estado do banco de dados. Essas restrições são derivadas de regras do minimundo que o banco de dados representa, conforme discutimos na Seção 1.6.8.

Nesta seção discutiremos as várias limitações nos dados que podem ser especificadas em um banco de dados relacional, na forma de restrições. As restrições em bancos de dados podem, geralmente, ser divididas em três categorias principais:

1. Restrições que são inerentes ao modelo de dado. Vamos chamá-las restrições inerentes baseadas **em modelo**.
2. Restrições que podem ser expressas diretamente nos esquemas do modelo de dado, normalmente por suas especificações em DDL (linguagem de definição de dado — *data definition language*, Seção 2.3.1). São conhecidas como **restrições** baseadas em esquema.

3. Restrições que *não podem* ser expressas diretamente nos esquemas do modelo de dado e, por isso, devem ser expressas e impostas pelos programas de aplicação. Chamamo-nas de **restrições baseadas em aplicação**.

As características das relações, que discutimos na Seção 5.1.2, são as restrições inerentes do modelo relacional e pertencem à primeira categoria; por exemplo, a restrição de que uma relação não pode ter tuplas repetidas é aquela inerente. As restrições que discutiremos nesta seção são da segunda categoria, daquelas que podem ser expressas no esquema do modelo relacional por meio de DDL. As restrições da terceira categoria são mais gerais e difíceis de expressar e impor dentro do modelo de dados e, por isso, são geralmente asseguradas por meio dos programas de aplicação.

Outra categoria importante de restrições são as *dependências de dados*, que incluem as *dependências funcionais* e as *multi-valoradas*. Elas são usadas, principalmente, para testar a 'excelência' do projeto de um banco de dados relacional e são utilizadas em um processo chamado *normalização*, discutido nos capítulos 10 e 11.

Discutiremos, agora, os tipos principais de restrições que podem ser expressas no modelo relacional — as restrições baseadas em esquema, da segunda categoria. Incluem as restrições de domínio, restrições de chave, restrições em *null*, restrições de integridade de entidade e restrições de integridade referencial.

5.2.1 Restrições de Domínio

As restrições de domínio especificam que, dentro de cada tupla, o valor de cada atributo A deve ser um valor atômico do domínio $\text{dom}(A)$. Já discutimos, na Seção 5.1.1, as formas pelas quais os domínios podem ser especificados. Os tipos de dados associados aos domínios incluem os tipos de dados numéricos padrões para inteiros (como inteiro curto, inteiro e inteiro longo) e números reais (ponto flutuante e flutuante de precisão dupla). Os caracteres, booleanos e as cadeias de caracteres de comprimento fixo e de comprimento variável também estão disponíveis, como data, hora, *timestamp* (marca de tempo) e, em alguns casos, os tipos de dado de moeda. Outros domínios possíveis podem ser descritos por um subconjunto de valores de um tipo de dado ou por um tipo de dado enumerado, por meio dos quais todos os possíveis valores serão, explicitamente, relacionados. Em vez de descrevê-los em detalhes aqui, discutiremos os tipos de dados oferecidos pelo padrão relacional SQL-99 na Seção 8.1.

5.2.2 Restrições de Chave e em Valores Null

Uma *relação* é definida como um *conjunto de tuplas*. Por definição, todos os elementos de um conjunto são distintos, por isso, todas as tuplas da relação também devem ser distintas. Isso significa que duas tuplas não podem ter a mesma combinação de valores para todos os seus atributos. Geralmente, há outros subconjuntos de atributos de um esquema de relação R com a propriedade de que duas tuplas, em qualquer estado de relação r de R , não tenham as mesmas combinações de valores para esses atributos. Suponha que indiquemos um desses subconjuntos de atributos por SK ; então, para quaisquer duas tuplas *distintas* t_1 e t_2 em um estado de relação r de R , teremos a restrição que

$t_1[SK] \neq t_2[SK]$

Qualquer um desses conjuntos de atributos SK é chamado *superchave* do esquema de relação R . Uma superchave SK especifica uma restrição de *unicidade*, na qual duas tuplas distintas, em qualquer estado r de R , não podem ter o mesmo valor para SK . Toda relação tem, pelo menos, uma superchave *default* — o conjunto de todos os seus atributos. Uma superchave pode ter os atributos redundantes, entretanto, um conceito mais vantajoso é o de uma *chave* que não apresenta redundância. Uma chave K de um esquema de relação R é uma superchave de R , com a propriedade adicional de, ao remover qualquer atributo A de K , o conjunto de atributos K restante não será mais uma superchave de R . Portanto, uma chave satisfaz duas restrições:

1. Duas tuplas distintas, em qualquer estado da relação, não podem ter valores idênticos para (todos) os atributos da chave.
2. Ela é uma *superchave mínima* — isto é, uma superchave da qual não podemos remover quaisquer atributos e ainda manter a restrição de unicidade garantida pela condição 1.

A primeira condição se aplica tanto às chaves como às superchaves. A segunda condição é requerida apenas para as chaves. Por exemplo, considere a relação $ALUNO$ da Figura 5.1. O conjunto de atributo $\{SSN\}$ é uma chave de $ALUNO$, porque duas tuplas de aluno não podem ter o mesmo valor para SSN . Todo conjunto de atributos que contiver SSN — por exemplo, $\{SSN, Nome, Idade\}$ — será uma superchave. Entretanto, a superchave $\{SSN, Nome, Idade\}$ não é uma chave de $ALUNO$, pois remover $Nome$ ou $Idade$, ou ambos, do conjunto, ainda nos deixa com uma superchave. Em geral, toda superchave, formada por um único atributo, também é uma chave. Uma chave com vários atributos exige que todos garantam a propriedade de unicidade.

O valor de um atributo-chave também pode ser usado para identificar, unicamente, cada tupla na relação. Por exemplo, o valor 305-61-2435 de SSN identifica unicamente a tupla correspondente a Benjamin Bayer na relação $ALUNO$. Observe que um conjunto de atributos constituindo uma chave é uma propriedade do esquema da relação; é uma restrição que se manteria em todos os estados válidos de relação desse esquema. Uma chave é determinada em função do significado dos atributos, e a propriedade *não varia no tempo*: ela deve continuar a ser mantida quando inserimos novas tuplas na relação. Por exemplo, não podemos e não deveríamos indicar o atributo $Nome$, da relação $ALUNO$ na Figura 5.1, como uma chave, pois é possível existirem dois alunos com nomes idênticos, a qualquer momento, em um estado válido.

CARRO	NumeroLicenca	NumeroChassi	Marca	Modelo	Ano
	Texas ABC-739	A69352	Ford	Mustang	96
	Flórida TVP-347	B43696	Oldsmobile	Cutlass	99
	Nova York MPO-22	X83554	Oldsmobile	Delta	95
	Califórnia 432-TFY	C43742	Mercedes	190-D	93
	Califórnia RSK-629	Y82935	Toyota	Camry	98
	Texas RSK-629	U028365	Jaguar	XJS	98

FIGURA 5.4 A relação $CARRO$ com duas chaves candidatas: $NumeroLicenca$ e $NumeroChassi$.

Em geral, um esquema de relação pode ter mais de uma chave. Nesse caso, cada uma das chaves é chamada *chave candidata*. Por exemplo, a relação $CARRO$, na Figura 5.4, tem duas chaves candidatas: $NumeroLicenca$ e $NumeroChassi$. É comum indicar uma das chaves candidatas como a chave primária da relação. Essa é a chave candidata cujos valores são usados para *identificar* tuplas na relação. Usaremos a convenção de que os atributos que formam a chave primária de um esquema de relação devam ser sublinhados, como mostrado na Figura 5.4. Observe que, quando um esquema de relação tem diversas chaves

8 Observe que SSN também é uma superchave.

9 Os nomes são, às vezes, usados como chaves, porém, algum artifício — como anexar um número ordinal — precisa ser empregado para distinguir os nomes idênticos.

96 Capítulo 5 O Modelo de Dados Relacional e as Restrições de um Banco de Dados Relacional

candidatas, a escolha de uma para se tornar a chave primária é arbitrária, entretanto, geralmente é melhor optar por uma chave primária com um atributo único ou com um número pequeno de atributos.

Outra restrição nos atributos especifica se valores *null* são ou não permitidos. Por exemplo, se toda tupla de ALUNO tem um valor válido, não nulo, para o atributo Nome, então Nome de ALUNO é forçado a ser **NOT NULL (não nulo)**.

5.2.3 Bancos de Dados Relacionais e Esquemas de um Banco de Dados Relacional

As definições e restrições que temos discutido até aqui se aplicam às relações únicas e seus atributos. Um banco de dados relacional contém, geralmente, muitas relações, com suas tuplas relacionadas de várias maneiras entre si. Nesta seção, definimos um banco de dados e um esquema de um banco de dados relacional. Um **esquema de um banco de dados relacional** S é um conjunto de esquemas de relação $S = \{R_1, R_2, \dots, R_m\}$ e um conjunto de **restrições de integridade IC**. Um **estado de um banco de dados relacional** DB de S é um conjunto dos estados da relação $DB = \{r_1, r_2, \dots, r_m\}$, de forma que cada r_i seja um estado de R_i , e de maneira que os estados da relação r_i satisfaçam as restrições de integridade especificadas em IC. A Figura 5.5 mostra um esquema de um banco de dados relacional que chamamos EMPRESA = (EMPREGADO, DEPARTAMENTO, DEPTO_LOCALIZACOES, PROJETO, TRABALHA_EM, DEPENDENTE). Os atributos sublinhados representam as chaves primárias. A Figura 5.6 mostra um estado do banco de dados relacional, correspondendo ao esquema EMPRESA. Usaremos esse esquema e estado de um banco de dados neste capítulo, e nos capítulos 6 a 9, para o desenvolvimento de consultas de exemplo nas diferentes linguagens relacionais. Quando nos referirmos a um banco de dados relacional incluímos, implicitamente, seu esquema e seu estado corrente. Um estado de um banco de dados que não obedece a todas as restrições de integridade é chamado **estado inválido**, e um estado que satisfaz todas as restrições em IC é conhecido como **estado válido**.

EMPREGADO

<u>PNOME</u>	<u>MINICIAL</u>	<u>UNOME</u>	<u>SSN</u>	<u>DATANASC</u>	<u>ENDEREÇO</u>	<u>SEXO</u>	<u>SALÁRIO</u>	<u>SUPERSSN</u>	<u>DNO</u>

DEPARTAMENTO

<u>DNOME</u>	<u>DNUMERO</u>	<u>GERSSN</u>	<u>GERDATAINICIO</u>

DEPTO LOCALIZAÇÕES

<u>DNUMERO</u>	<u>DLOCALIZACAO</u>

PROJETO

<u>PJNOME</u>	<u>PNUMERO</u>	<u>PLOCALIZACAO</u>	<u>DNUM</u>

<u>TRABALHA_</u>	<u>EM</u>
<u>ESSN</u>	<u>PNO</u>

DEPENDENTE

<u>ESSN</u>	<u>NOME DEPENDENTE</u>	<u>SEXO</u>	<u>DATANASC</u>	<u>PARENTESCO</u>

FIGURA 5.5 Diagrama para o esquema do banco de dados relacional EMPRESA.

Na Figura 5.5, o atributo DNUMERO em DEPARTAMENTO e DEPTO_LOCALIZACOES abriga o mesmo conceito do mundo real — o número de um dado departamento. Esse mesmo conceito é chamado DNO em EMPREGADO, e DNUM em PROJETO. Os atributos que representam o mesmo conceito do mundo real podem ou não ter nomes idênticos em diferentes relações. Alternativamente, os atributos que representam diferentes conceitos podem ter o mesmo nome em diferentes relações. Por exemplo, poderíamos ter usado o nome de atributo NOME para PJNOME de PROJETO, e DNOME de DEPARTAMENTO; nesse caso, teríamos dois atributos que compartilham o mesmo nome, mas que representam conceitos diferentes do mundo real — nomes de projeto e nomes de departamento.

10 Um *estado* de um banco de dados é, às vezes, chamado *instância* do banco de dados relacional. Entretanto, como mencionamos anteriormente, não usaremos o termo *instância*, uma vez que ele também se aplica a tuplas únicas.

Em algumas versões anteriores do modelo relacional, considerou-se que o mesmo conceito do mundo real, quando representado por um atributo, deveria ter nomes *idênticos* em todas as relações. Isso cria problemas quando o mesmo conceito do mundo real é usado em diferentes papéis (significados) na mesma relação. Por exemplo, o conceito de número do seguro social aparece duas vezes na relação EMPREGADO da Figura 5.5: uma no papel de número do seguro social do empregado e outra no papel de número do seguro social do supervisor. Nós lhes demos nomes de atributo distintos — SSN e SUPERSSN, respectivamente — de maneira a distinguir seus significados.

EMPREGADO	PNOME	MINICIAL	UNOME	SSN	DATANASC	ENDEREÇO	SEXO	SALÁRIO	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	<i>null</i>	1

DEPT_LOCALIZACOES

DEPARTAMENTO	DNOME	DNUMERO	GERSSN	GERDATAINICIO
	Pesquisa	5	333445555	1988-05-22
	Administração	4	987654321	1995-01-01
	Sede administrativa	1	888665555	1981-06-19

DNUMERO

DLOCALIZACAO

Houston

Bellaire

Sugarland

Houston

TRABALHAREM	ESSN	PNO	HORAS
	123456789	1	32.5
	123456789	2	75
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	200
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	<i>null</i>

PROJETO	PJNOME	PNUMERO	PLOCALIZACAO	DNUM
	ProdutoX	1	Bellaire	5
	ProdutoY	2	Sugarland	5
	ProdutoZ	3	Houston	5
	Automatização	10	Stafford	4
	Reorganização	20	Houston	1
	Novos Benefícios	30	Stafford	4

DEPENDENTE	ESSN	NOMEJ3EPENDENTE	SEXO	DATANASC	PARENTESCO
	333445555	Alice	F	1986-04-05	FILHA
	333445555	Theodore	M	1983-10-25	FILHO

333445555	Joy	F	1958-05-03	CÔNJUGE
987654321	Abner	M	1942-02-28	CÔNJUGE
123456789	Michael	M	1988-01-04	FILHO
123456789	Alice	F	1988-12-30	FILHA
123456789	Elizabeth	F	1967-05-05	CÔNJUGE

FIGURA 5.6 Um estado de um banco de dados possível para o esquema do banco de dados relacional EMPRESA.

Cada SGBD relacional deve ter uma linguagem de definição de dados (DDL) para a definição de um esquema de um banco de dados relacional. Os SGBDs atuais estão, em sua maior parte, usando o SQL com esse objetivo. Das seções 8.1 até 8.3, apresentaremos a DDL do SQL.

As restrições de integridade são especificadas no esquema do banco de dados, e espera-se que sejam mantidas em qualquer estado válido do banco de dados desse esquema. Além das restrições de domínio, chave e NOT NULL, outros dois tipos de restrições são considerados parte do modelo relacional: de integridade de entidade e de integridade referencial.

5.2.4 Integridade de Entidade, Integridade Referencial e Chaves Estrangeiras

A **restrição de integridade de entidade** estabelece que nenhum valor de chave primária pode ser *null*. Isso porque o valor da chave primária é usado para identificar as tuplas individuais em uma relação. Ter valores *null* para a chave primária implica não podermos identificar alguma tupla. Por exemplo, se duas ou mais tuplas tiverem *null* em suas chaves primárias, poderemos não ser capazes de distingui-las, se tentarmos fazer referência a elas por intermédio de outras relações.

As restrições de chave e as de integridade de entidade são especificadas na própria relação. A **restrição de integridade referencial** é classificada entre duas relações e é usada para manter a consistência entre as tuplas nas duas relações. Informalmente, a restrição de integridade referencial declara que uma tupla em uma relação, que faz referência a outra relação, deve se referir a uma *tupla existente* nessa relação. Por exemplo, na Figura 5.6, o atributo DNO de EMPREGADO fornece o número do departamento para o qual cada empregado trabalha, portanto, seu valor em toda tupla de EMPREGADO deve corresponder ao valor do DNUMERO de alguma tupla na relação DEPARTAMENTO.

Para definir integridade referencial mais formalmente definimos, primeiro, o conceito de uma *chave estrangeira*. As condições dadas a seguir para uma chave estrangeira especificam uma restrição de integridade referencial entre os dois esquemas de relação R_1 e R_2 . Um conjunto de atributos FK do esquema da relação R_1 é uma **chave estrangeira** de R_1 , que faz **referência** à relação R_2 , se ele satisfizer as duas regras seguintes:

1. Os atributos de FK têm o(s) mesmo(s) domínio(s) que os atributos da chave primária PK de R_2 ; diz-se que os atributos de FK fazem **referência** ou se **referem** à relação R_2 .
2. Um valor de FK em uma tupla t_1 do estado corrente R_1 ocorre como um valor de PK para alguma tupla t_2 no estado corrente R_2 ou é *null*. No caso anterior, temos $t_1[FK] = t_2[PK]$, e dizemos que a tupla t_1 faz **referência** ou se **refere** à tupla t_2 .

Nessa definição, R_1 é chamada **relação referência**, e R_2 é a **relação referida**. Se essas duas condições forem asseguradas, uma **restrição de integridade referencial** de R_1 para R_2 é dita garantida. Em um banco de dados com muitas relações, geralmente há muitas restrições de integridade referencial.

Para especificar essas restrições devemos, primeiro, ter um entendimento claro do significado, ou do papel, que cada conjunto de atributos desempenha nos vários esquemas de relação do banco de dados. As restrições de integridade referencial surgem dos relacionamentos entre as entidades representadas em esquemas de relação. Por exemplo, considere o banco de dados mostrado na Figura 5.6. Na relação EMPREGADO, o atributo DNO refere-se ao departamento para o qual um empregado trabalha, portanto, indicamos DNO como chave estrangeira de EMPREGADO, fazendo referência à relação DEPARTAMENTO. Isso significa que um valor de DNO em qualquer tupla t_1 da relação EMPREGADO deve corresponder a um valor da chave primária de DEPARTAMENTO — o atributo DNUMERO — em alguma tupla t_2 da relação DEPARTAMENTO, ou o valor de DNO *pode ser null* se o empregado não pertencer a um departamento. Na Figura 5.6, a tupla para o empregado 'John Smith' faz referência à tupla para o departamento 'Pesquisa', indicando que 'John Smith' trabalha para esse departamento.

Observe que uma chave estrangeira pode *se referir à sua própria relação*. Por exemplo, o atributo SUPERSSN em EMPREGADO refere-se ao supervisor de um empregado; esse é outro empregado, representado por uma tupla na relação EMPREGADO. Portanto, SUPERSSN é uma chave estrangeira que faz referência, ela mesma, à relação EMPREGADO. Na Figura 5.6, a tupla do empregado 'John Smith' faz referência à tupla do empregado 'Franklin Wong', indicando que 'Franklin Wong' é o supervisor de 'John Smith'.

Podemos *exibir diagramaticamente as restrições de integridade referencial* puxando um arco direto de cada chave estrangeira à relação que ela faz referência. Para maior clareza, a seta deve apontar para a chave primária da relação referida. A Figura 5.7 mostra o esquema da Figura 5.5 com as restrições de integridade referencial exibidas dessa maneira.

Todas as restrições de integridade deveriam ser especificadas no esquema do banco de dados relacional, caso queiramos impor essas restrições aos estados do banco de dados. Portanto, a DDL possui recursos para especificar os vários tipos de restrições e, assim, o SGBD pode, automaticamente, garanti-las. A maioria dos SGBDs relacionais suporta restrições de chave e de integridade de entidade, e providenciam suporte para a integridade referencial. Essas restrições são especificadas como parte da definição de dados.

5.2 Restrições do Modelo Relacional e Esquemas de um Banco de Dados Relacional 99

EMPREGADO

PNO ME MINICIAL UNOME SSN DATANASC ENDEREÇO SEXO SALÁRIO SUPERSSN DNO
 DNO ME DNUMERO GERSSN
DEPTO LOCALIZACOES
 GERDATAINICIO
 DNUMERO DLOCALIZACAO

PLOCALIZACAO

DNUM

TRABALHA EM

ESSN / PNO HORAS

DEPENDENTE

ESSN NOME_DEPENDENTE SEXO DATANASC PARENTESCO

FIGURA 5.7 Restrições de integridade referencial exibidas no esquema de um banco de dados relacional empresa.

5.2.5 Outros Tipos de Restrições

As restrições de integridade anteriores não incluem uma grande classe de restrições genéricas chamadas, às vezes, *restrições de integridade semântica*, que podem ser especificadas e impostas em um banco de dados relacional. Os exemplos dessas restrições são 'o salário de um empregado não deveria exceder o do supervisor do empregado' e 'o número máximo de horas que um empregado pode trabalhar por semana, em todos os projetos, é 56'. Essas restrições podem ser especificadas e impostas dentro dos programas de aplicação que atualizam o banco de dados ou pelo uso de uma **linguagem de especificação de restrição** de propósito geral. Os mecanismos conhecidos como **gatilhos e asserções** podem ser usados. Na SQL-99, uma declaração CREATE ASSERTION é usada com esse objetivo (capítulos 8 e 9). A imposição desse tipo de restrição é mais comum dentro dos programas de aplicação que pelo uso de linguagens de especificação de restrição, pois essas últimas são difíceis e complexas de serem usadas corretamente, conforme discutiremos na Seção 24.1.

Outro tipo de restrição é a de *dependência funcional*, que estabelece um relacionamento funcional entre dois conjuntos de atributos X e Y. Essa restrição especifica que o valor de X determina o valor de Y em todos os estados de uma relação — isso é indicado como uma dependência funcional $X \rightarrow Y$. Usaremos as dependências funcionais e outros tipos de dependências, nos capítulos 10 e 11, como ferramentas para avaliar a qualidade de projetos relacionais e para 'normalizar' as relações, de modo a melhorar sua qualidade.

Os tipos de restrições discutidos até aqui podem ser chamados **restrições de estado**, porque definem as restrições que um *estado válido* do banco de dados deve satisfazer. Outro tipo de restrição, conhecido como **restrições de transição**, pode ser estabelecido para lidar com as mudanças de estado no banco de dados. Um exemplo de uma restrição de transição é: 'o salário de um empregado pode ser apenas aumentado'. Essas restrições são impostas pelos programas de aplicação ou especificadas usando-se regras ativas e gatilhos, como discutiremos na Seção 24.1.

11 As restrições de estado às vezes são chamadas *restrições estáticas*, e as restrições de transição, em algumas ocasiões, são conhecidas como *restrições dinâmicas*.

5.3 OPERAÇÕES DE ATUALIZAÇÃO E TRATAMENTO DE VIOLAÇÕES DE RESTRIÇÃO

As operações do modelo relacional podem ser categorizadas em *recuperações* e *atualizações*. As operações da álgebra relacional, que podem ser usadas para especificar as recuperações, serão discutidas em detalhes no Capítulo 6. Uma expressão da álgebra relacional forma uma nova relação depois de aplicar um número de operadores algébricos a um conjunto existente de relações; seu uso principal é para consultar um banco de dados. O usuário formula uma consulta que especifica o dado de interesse e uma nova relação é formada pela aplicação de operadores relacionais para recuperar esse dado. Essa relação se torna a resposta à consulta do usuário. O Capítulo 6 também introduz a linguagem chamada cálculo relacional, que é usada para definir, declarativamente, a nova relação, sem dar uma ordem específica das operações.

Nesta seção concentramo-nos nas operações para a **modificação** ou **atualização** no banco de dados. Há três operações básicas de atualização em relações: *insert*, *delete* e *modify*. **Insert** é usada para inserir uma nova tupla ou tuplas em uma relação. **Delete** é empregada para remover as tuplas. E **Update** (ou **Modify**) é utilizada para mudar os valores de alguns atributos em tuplas existentes. Quando essas operações são aplicadas, as restrições de integridade especificadas no esquema do banco de dados relacional não devem ser violadas. Nesta seção, discutiremos os tipos de restrições que podem ser violadas em cada operação de atualização e os tipos de ações que devem ser tomadas se uma atualização causar uma violação. Usamos o banco de dados mostrado na Figura 5.6 para exemplos e discutimos apenas as restrições de chave, de integridade de entidade, bem como as restrições de integridade referencial, mostradas na Figura 5.7. Para cada tipo de atualização damos algumas operações de exemplo e discutimos as restrições que cada operação pode violar.

5.3.1 A Operação *Insert*

A operação **Insert** fornece uma lista de valores de atributos para uma nova tupla *t*, que é inserida em uma relação *R*. **Insert** pode violar qualquer um dos quatro tipos de restrições discutidas na seção anterior. As restrições de domínio podem ser violadas se um valor de atributo que é dado não pertencer ao domínio correspondente. Já as restrições de chave podem ser violadas se um valor de chave, em uma nova tupla *t*, já existir em outra tupla da relação *r(R)*. A integridade de entidade pode ser violada se a chave primária da nova tupla *t* for *null*. A integridade referencial pode ser violada se o valor de qualquer chave estrangeira em *t* referir-se a uma tupla que não existe na relação referida. Aqui estão alguns exemplos para ilustrar a discussão.

1. Inserir <'Cecilia', 'P', 'Kolonsky', *null*, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, *null*, 4' > em EMPREGADO.

• Essa inserção viola a restrição de integridade de entidade (*null* para a chave primária SSN), assim, ela será rejeitada.

2. Inserir <'Alicia', 'J', 'Zelava', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4> em EMPREGADO.

• Essa inserção viola a restrição de chave, porque outra tupla com o mesmo valor de SSN já existe na relação EMPREGADO, então ela será rejeitada.

3. Inserir <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windswept, Katy, TX', F, 28000, '987654321', 7> em EMPREGADO.

• Essa inserção viola a restrição de integridade referencial especificada em DNO, porque não existe tupla de DEPARTAMENTO com DNUMERO = 7.

4. Inserir <'Cecilia', 'P', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, *null*, 4> em EMPREGADO.

• Essa inserção satisfaz todas as restrições, assim, ela é aceitável.

Se uma inserção violar uma ou mais restrições, a opção *default* é *rejeitar a inserção*. Nesse caso, seria útil se o SGBD explicasse ao usuário por que a inserção foi rejeitada. Outra opção é tentar *corrigir a razão para a rejeição da inserção*, mas isso não é comum para as violações causadas por *Insert*; ela é preferencialmente usada com mais frequência na correção de violações para *Delete* e *Update*. Na operação 1 acima, o SGBD poderia solicitar ao usuário um valor para SSN, e poderia aceitar a inserção se um valor válido para SSN fosse fornecido. Na operação 3, o SGBD também poderia solicitar ao usuário que mudasse o valor de DNO para um valor válido (ou determiná-lo como *null*), ou poderia pedir ao usuário que inserisse uma tupla de DEPARTAMENTO com DNUMERO = 7, aceitando a primeira inserção somente depois que essa operação fosse aceita. Observe que, nesse último caso, a violação de inserção pode atingir a relação EMPREGADO após a **cascata**, caso o usuário tente inserir uma tupla para o departamento 7 com um valor de GERSSN que não exista na relação EMPREGADO.

5.3.2 A Operação *Delete*

A operação *Delete* pode violar apenas a integridade referencial se a tupla removida for referida por chaves estrangeiras de outras tuplas no banco de dados. Para especificar a remoção, uma condição nos atributos da relação seleciona a tupla (ou tuplas) a ser removida. Eis alguns exemplos.

1. Remover a tupla TRABALHA_EM com ESSN = '999887777' e PNO = 10.
 - Essa remoção é aceitável.
2. Remover a tupla EMPREGADO com SSN = '999887777'.
 - Essa remoção não é aceitável, porque há tuplas em TRABALHA_EM que se referem a essa tupla. Portanto, se a tupla for removida, vão resultar violações de integridade referencial.
3. Remover a tupla EMPREGADO com SSN = '333445555'.
 - Essa remoção resultará em violações de integridade referencial piores ainda, porque a tupla envolvida é referida por tuplas das relações EMPREGADO, DEPARTAMENTO, TRABALHA_EM e DEPENDENTE.

Há diversas opções disponíveis caso uma operação de remoção cause violação. A primeira opção é *rejeitar a remoção*. A segunda é a *remoção em cascata (ou propagação)*, eliminando as tuplas que fazem referência à tupla que está sendo removida. Por exemplo, na operação 2, o SGBD poderia remover automaticamente as tuplas atingidas de TRABALHA_EM com SSN = '999887777'. Uma terceira opção é *modificar os valores dos atributos de referência* que causam a violação; cada um desses valores ou seria determinado por *null* ou modificado para referir outra tupla válida. Observe que, se um atributo de referência que cause violação fizer parte da chave primária, ele não poderá ser determinado como *null*; de outra maneira, ele violaria a integridade da entidade.

As combinações dessas três opções também são possíveis. Por exemplo, para evitar que a operação 3 cause uma violação, o SGBD pode remover automaticamente todas as tuplas de TRABALHA_EM e DEPENDENTE com ESSN = '333445555'. As tuplas em EMPREGADO, com SUPERSSN = '333445555', e a tupla em DEPARTAMENTO, com GERSSN = '333445555', podem ter seus valores de SUPERSSN e GERSSN mudados para outros valores válidos ou para *null*. Embora faça sentido remover automaticamente as tuplas de TRABALHA_EM e DEPENDENTE, que se referem a uma tupla de EMPREGADO, não deve fazer sentido remover outras tuplas de EMPREGADO ou uma tupla de DEPARTAMENTO.

Em geral, quando uma restrição de integridade referencial é especificada na DDL, o SGBD permitirá ao usuário *especificar quais opções* se aplicam no caso de uma violação de restrição. Discutiremos como especificar essas opções na DDL do SQL-99 no Capítulo 8.

5.3.3 A Operação *Update*

A operação *Update* (ou *Modify*) é usada para mudar os valores de um ou mais atributos em uma tupla (ou tuplas) de alguma relação R. É necessário especificar uma condição nos atributos da relação para selecionar a tupla (ou tuplas) a ser modificada. Aqui estão alguns exemplos.

1. Modificar o SALÁRIO da tupla de EMPREGADO com SSN = '999887777' para 28000.
 - Aceitável.
2. Modificar o DNO da tupla de EMPREGADO com SSN = '999887777' para 1.
 - Aceitável.
3. Modificar o DNO da tupla de EMPREGADO com SSN = '999887777' para 7.
 - Inaceitável, porque viola a integridade referencial.
4. Modificar o SSN da tupla de EMPREGADO com SSN = '999887777' para '987654321'.
 - Inaceitável, pois viola as restrições de chave primária e as de integridade referencial.

Mudar um atributo que não é nem uma chave primária nem uma chave estrangeira geralmente não causa problemas; o SGBD precisa apenas verificar e confirmar que o novo valor é do tipo de dado e de domínio correto. Modificar um valor de chave primária é similar a remover uma tupla e inserir outra em seu lugar, porque usamos a chave primária para identificar as tuplas. Portanto, as questões discutidas anteriormente, nas Seções 5.3.1 (*insert*) e 5.3.2 (*Delete*), entram em execução. Se um atributo de chave estrangeira for modificado, o SGBD deverá se certificar de que o novo valor refere-se a uma tupla existente na relação referida (ou é *null*). Existem opções similares para tratar as violações de integridade referencial causadas por *Update*, como aquelas opções discutidas para a operação *Delete*. De fato, quando uma restrição de integridade referencial é especificada na DDL, o SGBD permitirá ao usuário escolher as opções distintas para tratar uma violação causada por *Delete* e uma causada por *Update* (Seção 8.2).

5.4 RESUMO

Neste capítulo apresentamos os conceitos de modelagem, as estruturas de dados e as restrições fornecidas pelo modelo relacional de dados. Começamos introduzindo o conceito de domínios, os atributos e as tuplas. Então definimos um esquema de relação como uma lista de atributos que descrevem a estrutura de uma relação. Uma relação, ou estado de relação, é um conjunto de tuplas que se amoldam ao esquema.

Diversas características diferenciam as relações de tabelas ou arquivos comuns. A primeira é que as tuplas em uma relação não estão ordenadas. A segunda envolve a ordenação de atributos em um esquema de relação e a correspondente ordenação de valores dentro de uma tupla. Demos uma definição alternativa de relação que não requer essas duas ordenações, mas continuamos a usar, por conveniência, a primeira definição, que exige que os valores dos atributos e as tuplas sejam ordenados. Discutimos, então, os valores em tuplas e introduzimos os valores *null* para representar uma informação ausente ou desconhecida.

Depois, classificamos as restrições de um banco de dados em: restrições inerentes baseadas em modelo, restrições baseadas em esquema e restrições baseadas em aplicação. Discutimos, então, as restrições de esquema relativas ao modelo relacional, começando por restrições de domínio, depois, restrições de chave, incluindo os conceitos de superchave, chave candidata e chave primária, e as restrições NOT NULL em atributos. Definimos, em seguida, os bancos de dados relacionais e esquemas de banco de dados relacional. As restrições relacionais adicionais incluem a restrição de integridade de entidade, que proíbe os atributos de chave primária de serem *null*. Foi então descrita a inter-relação de restrição de integridade referencial, que é usada para manter a consistência de referências entre as tuplas de diferentes relações.

As operações de modificação no modelo relacional são *Insert*, *Delete* e *Update*. Cada operação pode violar certos tipos de restrição. Essas operações foram discutidas na Seção 5.3. Sempre que uma operação for aplicada, o estado do banco de dados após a operação deverá ser verificado para garantir que nenhuma restrição tenha sido violada.

Questões de Revisão

- 5.1. Defina os seguintes termos: *domínio*, *atributo*, *n-tupla*, *esquema de relação*, *estado de relação*, *grau de uma relação*, *esquema de um banco de dados relacional*, *estado de um banco de dados relacional*.
- 5.2. Por que as tuplas não são ordenadas em uma relação?
- 5.3. Por que as tuplas repetidas não são permitidas em uma relação? 5.4- Qual é a diferença entre uma chave e uma superchave?
- 5.5. Por que designamos uma das chaves candidatas de uma relação para ser uma chave primária?
- 5.6. Discuta as características das relações que as fazem diferentes de tabelas e arquivos comuns.
- 5.7. Discuta as várias razões que levam à ocorrência de valores *null* nas relações.
- 5.8. Discuta as restrições de integridade de entidade e de integridade referencial. Por que cada uma é considerada importante?
- 5.9. Defina *chave estrangeira*. Para que esse conceito é usado?

Exercícios

- 5.10. Suponha que cada uma das seguintes operações de atualização seja aplicada diretamente ao estado do banco de dados mostrado na Figura 5.6. Discuta todas as restrições de integridade violadas em cada operação, se houver alguma, e os diferentes mecanismos para impor essas restrições.
 - a. Insira <'Robert', 'F', 'Scott', '943775543', '1952-06-21', '2365 Newcastle Rd, Bellaire, TX', M, 58000, '888665555', 1> em EMPREGADO.
 - b. Insira <'ProdutoA', 4, 'Bellaire', 2> em PROJETO.
 - c. Insira <'Produção', 4, '943775543', '1998-10-01'> em DEPARTAMENTO.
 - d. Insira <'677678989', *null*, '40,0'> em TRABALHA_EM.
 - e. Insira <'453453453', 'John', M, '1970-12-12', 'CONJUGE'> em DEPENDENTE.
 - f. Remova as tuplas de TRABALHA_EM com ESSN = '333445555'.
 - g. Remova a tupla de EMPREGADO com SSN = '987654321'.
 - h. Remova a tupla de PROJETO com PJNOME = 'ProdutoX'.
 - i. Modifique o GERSSN e GERDATAINICIO da tupla de DEPARTAMENTO com DNUMERO = 5 para '123456789' e '1999-01-10', respectivamente.
 - j. Modifique o atributo SUPERSSN da tupla de EMPREGADO com SSN = '999887777' para '943775543'.
 - k. Modifique o atributo HORAS da tupla de TRABALHA_EM com ESSN = '999887777' e PNO = 10 para '5,0'.

5.4 Resumo

103

5.11. Considere o esquema de um banco de dados relacional COMPANHIA AÉREA, mostrado na Figura 5.8, que descreve um banco de dados para as informações de vôos de uma companhia aérea. Cada VOO é identificado por um NUMERO de vôo e consiste em um ou mais TRECHO_VOO com NUMERO_TRECHO iguais a 1, 2, 3, e assim por diante. Cada vôo tem programados os horários de chegada e partida e os aeroportos, e possui muitas INSTANCIAS_TRECHO para cada DATA na qual o vôo viaja. As ocorrências de PASSAGEM são mantidas para cada vôo. Para cada instância de trecho são mantidas RESERVAS_POLTRONA, assim como as ocorrências de AVIAO usadas no trecho, os horários de chegada e partida e os aeroportos. Um AVIAO é identificado por um ID_AVIAO, e é de um TIPO_AVIAO em particular. PODE_POUSAR relaciona os TIPOS_AVIAO aos AEROPORTOS nos quais pode pousar. Um AEROPORTO é identificado por um CODIGO_AEROPORTO. Considere a seguinte atualização no banco de dados de uma COMPANHIA AÉREA: entrar com uma reserva em um vôo em particular ou um trecho de vôo em uma determinada data.

- Forneça as operações para essa atualização.
- Quais tipos de restrições poderiam ser esperadas para a verificação?
- Quais destas são restrições de chave, de integridade de entidade e de integridade referencial e quais não são?
- Especifique todas as restrições de integridade referencial envolvidas no esquema mostrado na Figura 5.8.

AEROPORTO

CODIGO_AEROPORTO	NOME	CIDADE	ESTADO
------------------	------	--------	--------

VOO

NUMERO	COMPANHIA AÉREA	DIA SEMANA
--------	-----------------	------------

TRECHO_VOO

NUMERO_VOO NUMERO_TRECHO CODIGO_AEROPORTO_PARTIDA HORA_PARTIDA_PROGRAMADACODIGO_AEROPORTO_CHEGADA HORA_CHEGADA_PROGRAMADA

INSTANCIA_TRECHO

NUMERO_VOO

NUMERO_TRECHO

DATA

NUMERO_POLTRONAS_DISPONIVEIS

ID_AVIAO

CODIGO_AEROPORTO_PARTIDA HORA_PARTIDA CODIGO_AEROPORTO_CHEGADA HORA_CHEGADA

PASSAGEM

NUMERO_VOO CODIGO_PASSAGEM VALOR RESTRIÇÕES

TIPO_AVIAO

NOME TIPO MAX_POLTRONAS EMPRESA

PODE_POUSAR

NOME TIPO AVIAO CÓDIGO AEROPORTO

AVIAO

ID_AVIAO

NUMERO_TOTAL_DE_POLTRONAS TIPO_AVIAO

RESERVA_POLTRONA

NUMERO_VOO NUMERO_TRECHO DATA NUMERO_POLTRONA NOME_CLIENTE FONE_CLIENTE

FIGURA 5.8 O esquema de um banco de dados relacional COMPANHIA AÉREA.

5.12. Considere a relação AULA(Curso#, Disciplina#, NomeInstrutor, Semestre, CodigoEdificio, Sala#, HorarioPeriodo, DiasUteis, CreditoHoras). Ela representa as aulas ministradas em uma universidade em uma única Disciplina*. Identifique quais você acha que seriam as várias chaves candidatas e escreva, com suas próprias palavras, as restrições sob as quais cada chave candidata seria válida.

104 Capítulo 5 O Modelo de Dados Relacional e as Restrições de um Banco de Dados Relacional

5.13. Considere as seis seguintes relações de uma aplicação de um banco de dados para o processamento de pedidos de uma empresa:

CUENTE(C1#, Cnome, Cidade).

PEDiDo(Pedido#, Pdata, Cust#, Tot_Ped).

ITEM PEDiDO(Pedido#, Item#, Qtde).

ITEM(Item#, PreCo_unit).

REMESSA(Pedido#, Armazem#, Data_rem).

DEPOSITO(Deposito#, Cidade).

Aqui, Tot_Ped refere-se ao total em dólares de um pedido; Pdata é a data em que o pedido foi feito; e Data_rem, a data em que é feita a remessa de um pedido de um dado depósito. Pressuponha que um pedido possa ser remetido a partir de diversos depósitos. Especifique as chaves estrangeiras para esse esquema, explicando quaisquer suposições que você faça.

5.14. Considere as seguintes relações de um banco de dados para o controle de viagens dos vendedores em um escritório de vendas:

VENDEDOR(SSN, Nome, Ano_Inicio, Num_Depto).

VIAGEM(SSN, Cidade_Origem, Cidade_Destino, Data_Partida, Data_Retorno, ID Viagem, Conta#, Total).

Especifique as chaves estrangeiras para esse esquema declarando quaisquer suposições que você faça.

5.15. Considere as seguintes relações para um banco de dados para o controle de alunos inscritos em cursos e os livros adotados para cada curso:

ALUNO(SSN, Nome, Curso_Hab, DataNasc).

CURSo(Curso#, Cnome, Depto).

INSCRICA0(SSN, Curso#, Trimestre, Nota).

ADOCAR HVROS(Curso#, Trimestre, ISBN_Livro).

TEXTOSBN Livro, Titulo_Livro, Editora, Autor).

Especifique as chaves estrangeiras para esse esquema declarando quaisquer suposições que você faça.

5.16. Considere as seguintes relações para um banco de dados para o controle de vendas de automóveis em uma concessionária (Opcional se refere a alguns equipamentos opcionais instalados em um automóvel):

CARRO(Num Serie, Modelo, Fabricante, Preço).

OPCiONAL(Num Serie, Nome Opcao, Preço).

VENDASGD Vendedor, Num Serie, Data, Preço_Venda).

VENDEDOR(VD Vendedor, Nome, Fone).

Primeiro, especifique as chaves estrangeiras para esse esquema, declarando quaisquer suposições que você faça. Depois, preencha as relações com algumas tuplas de exemplo e então dê um exemplo de inserção nas relações VENDAS e VENDEDOR que *virole* as restrições de integridade referencial, e outra que não o faça.

Bibliografia Seleccionada

O modelo relacional foi introduzido por Codd (1970) em um artigo clássico. Codd também introduziu a álgebra relacional e estabeleceu os fundamentos teóricos para o modelo relacional em uma série de artigos (Codd, 1971, 1972, 1972a e 1974); ele recebeu, mais tarde, o Prêmio de Turing, a mais alta honra da ACM, por seu trabalho no modelo relacional. Em um artigo posterior, Codd (1979) discutiu o modelo relacional estendido para incorporar mais metadados e semânticas sobre as relações; ele também propôs uma lógica de três valores para tratar as incertezas nas relações e incorporar NULLS na álgebra relacional. O modelo resultante é conhecido como um RM/T. Childs (1968) havia utilizado, anteriormente, a teoria de conjunto para

5.4 Resumo

105

modelar o banco de dados. Depois, Codd (1990) publicou um livro examinando mais de 300 características do modelo de dados relacional e sistemas de banco de dados.

Desde o trabalho pioneiro de Codd, muitas pesquisas têm sido conduzidas em vários aspectos do modelo relacional. Todd (1976) descreve um SGBD experimental chamado PRTV que implementa, diretamente, as operações de álgebra relacional. Schmidt e Swenson (1975) introduzem semânticas adicionais dentro do modelo relacional pela classificação dos tipos diferentes de relações. O modelo entidade-relacionamento de Chen (1976), que é discutido no Capítulo 3, é um meio para comunicar as semânticas do mundo real para um banco de dados relacional em nível conceitual. Wiederhold e Elmasri (1979) introduzem vários tipos de conexões entre as relações para incorporar suas restrições. As extensões do modelo relacional são discutidas no Capítulo 24. As notas bibliográficas adicionais para outros aspectos do modelo relacional e suas linguagens, sistemas, extensões e teoria são fornecidas nos capítulos de

6 a 11, 15, 16, 17, e 22 a 25.

6

A Álgebra Relacional e o Cálculo Relacional

Neste capítulo discutiremos as duas linguagens formais do modelo relacional: a álgebra relacional e o cálculo relacional. Conforme discutimos no Capítulo 2, um modelo de dados inclui um conjunto de operações para manipular o banco de dados, além dos conceitos de modelo de dados para a definição das restrições e estrutura do banco de dados. O conjunto básico de operações para o modelo relacional é a **álgebra relacional**. Essas operações permitem a um usuário especificar as solicitações básicas de recuperação. O resultado de uma recuperação será uma nova relação, que pode ter sido formada de uma ou mais relações. As operações de álgebra produzem, assim, novas relações, que podem ser manipuladas, adiante, usando-se as operações da mesma álgebra. Uma sequência de operações de álgebra relacional forma uma **expressão de álgebra relacional** cujos resultados também serão uma relação que representa o resultado de uma consulta de banco de dados (ou solicitação de recuperação).

A álgebra relacional é muito importante por diversas razões. Primeira, porque provê um fundamento formal para operações do modelo relacional. Segunda, e talvez a mais importante, porque é usada como uma base para implementar e otimizar as consultas em sistemas de gerenciadores de banco de dados relacional (SGBDRs), conforme discutimos na Parte IV deste livro. Terceira, alguns de seus conceitos são incorporados na linguagem de consulta-padrão SQL para os SGBDRs.

Enquanto a álgebra define um conjunto de operações para o modelo relacional, o **cálculo relacional** provê uma notação declarativa de nível superior para a especificação de consultas relacionais. Uma expressão de cálculo relacional cria uma nova relação, que é especificada em termos de variáveis que abrangem as linhas das relações armazenadas no banco de dados (em cálculos de tuplas) ou as colunas das relações armazenadas (em cálculo de domínio). Em uma expressão de cálculo, *não há ordem nas operações* para especificar como recuperar o resultado de uma consulta — uma expressão de cálculo especifica apenas qual informação o resultado deveria conter. Essa é a principal característica de distinção entre a álgebra relacional e o cálculo relacional. O cálculo relacional é importante porque tem uma sólida base na lógica matemática e porque a SQL (*standard query language* — linguagem de consulta-padrão) para os SGBDRs possui muitos de seus fundamentos no cálculo relacional de tupla.

A álgebra relacional freqüentemente é considerada como uma parte do modelo relacional de dados e suas operações podem ser divididas em dois grupos. Um grupo inclui um conjunto de operações da teoria de conjunto matemática — essas operações são aplicadas porque cada relação é definida como um conjunto de tuplas no modelo relacional formal. Os conjuntos de operações incluem UNIÃO (UNION), INTERSEÇÃO (INTERSECTION), DIFERENÇA DE CONJUNTO (SET DIFFERENCE) e PRODUTO CARTESIANO (CROSS PRODUCT). O outro grupo consiste em operações desenvolvidas especificamente para os bancos de dados relacionais — estas incluem SELEÇÃO (SELECT), PROJEÇÃO (PROJECT) e JUNÇÃO (JOIN), entre outras. Descreveremos, primeiro, na Seção 6.1, as operações SELEÇÃO e PROJEÇÃO, porque elas são **operações unárias** que atuam em relações

1 A SQL é baseada em cálculo relacional de tupla, mas também incorpora algumas das operações de álgebra relacional e suas extensões, conforme veremos nos capítulos 8 e 9.

6.1 Operações Relacionais Unárias: SELEÇÃO (SELECT) e PROJEÇÃO (PROJECT) 107

únicas. Discutiremos depois, na Seção 6.2, as operações de conjunto. Na Seção 6.3 abordaremos as JUNÇÕES e outras **operações binárias** complexas, que realizam duas tabelas. O banco de dados relacional EMPRESA, mostrado na Figura 5.6, será usado em nossos exemplos.

Algumas solicitações comuns de banco de dados não podem ser executadas com as operações originais da álgebra relacional, então, operações adicionais foram criadas para expressar essas solicitações. Estas incluem as **funções de agregação**, que são as operações que podem *sumarizar* os dados de tabelas, bem como os tipos adicionais de operações de JUNÇÃO e UNIÃO. Essas operações foram adicionadas à álgebra relacional original por causa da sua importância para muitas aplicações de banco de dados e serão descritas na Seção 6.4. Daremos exemplos de consultas de especificação que usam operações relacionais na Seção 6.5. Algumas dessas consultas são usadas nos capítulos subsequentes para ilustrar várias linguagens.

Nas seções 6.6 e 6.7, descreveremos a outra importante linguagem formal para os bancos de dados relacionais, o **cálculo relacional**. Há duas variações de cálculo relacional. O cálculo relacional de tupla será descrito na Seção 6.6, e o cálculo relacional de domínio, na Seção 6.7. Algumas das construções SQL discutidas no Capítulo 8 são baseadas em cálculo relacional de tupla. O cálculo relacional é uma linguagem formal, baseada em um ramo da lógica matemática chamada cálculo de predicado. No cálculo relacional de tupla, as variáveis abrangem as tuplas, enquanto no cálculo relacional de domínio, as variáveis englobam os domínios (valores) dos atributos. No Apêndice D damos uma avaliação da linguagem QBE (*Query-By-Example* — Consulta-Por-Exemplo), que é uma linguagem relacional gráfica amigável, baseada no cálculo relacional de domínio. A Seção 6.8 resume o capítulo.

Para o leitor interessado em uma introdução menos detalhada em linguagens relacionais formais, as seções 6.4, 6.6 e 6.7 podem ser desconsideradas.

6.1 OPERAÇÕES RELACIONAIS UNÁRIAS: SELEÇÃO (SELECT) E PROJEÇÃO (PROJECT)

6.1.1 A Operação SELEÇÃO

A operação SELEÇÃO é usada para selecionar um *subconjunto* de tuplas de uma relação que satisfaça uma **condição de seleção**. Uma operação que pode ser considerada SELEÇÃO é um *filtro* que mantém apenas aquelas tuplas que satisfaçam uma condição de qualificação. A operação SELEÇÃO também pode ser visualizada como um *particionamento horizontal* da relação em dois conjuntos de tuplas — aquelas tuplas que satisfazem a condição e são selecionadas, e as tuplas que não satisfazem a condição e são descartadas. Por exemplo, para selecionar as tuplas de EMPREGADO cujo departamento é 4, ou aquelas cujo salário é superior a 30 mil dólares, podemos especificar, individualmente, cada uma dessas duas condições com uma operação SELEÇÃO, conforme segue:

(XDN0=4 (EMPREGADO) aSALARIO > 30000 (EMPREGADO))

Em geral, a operação SELEÇÃO é indicada por

CT<condicao de selecao>(R)

em que o símbolo CT (sigma) é usado para indicar o operador SELEÇÃO e a condição de seleção é uma expressão booleana, especificada nos atributos da relação *R*. Observe que *R* é, geralmente, uma *expressão de álgebra relacional* cujo resultado é uma relação — a mais simples delas seria exatamente o nome de uma relação do banco de dados. A relação resultante da operação SELEÇÃO tem os *mesmos atributos* que *R*.

A expressão booleana especificada em <condicao de selecao> é composta por um número de **cláusulas** da forma

<nome do atributo> <op de comparacao> <valor da constante>,

ou

<nome do atributo> <op de comparacao> <nome do atributo>

2 Neste capítulo não se pressupõe nenhuma familiaridade com o cálculo de predicado de primeira ordem — que trata das variáveis e valores quantificados.

108 Capítulo 6 A Álgebra Relacional e o Cálculo Relacional

em que <nome do atributo> é o nome de um atributo de R; <op de comparação> corresponde, normalmente, a um dos operadores $\{=, <, <=, >, >=, \wedge\}$; e o <valor da constante> refere-se a um valor constante do domínio do atributo. As cláusulas podem ser conectadas arbitrariamente pelos operadores booleanos AND, OR e NOT para formar uma condição de seleção geral. Por exemplo, para selecionar as tuplas de todos os empregados que trabalham no departamento 4 e recebem acima de 25 mil dólares por ano, ou trabalham no departamento 5 e recebem acima de 30 mil dólares, podemos especificar a seguinte operação SELEÇÃO:

$\sigma_{(DNO=4 \text{ AND SALARIO}>25000) \text{ OR } (DNO=5 \text{ AND SALARIO}>30000)}(EMPREGADO)$

O resultado é mostrado na Figura 6.1a.

Observe que os operadores de comparação no conjunto $\{=, <, <=, >, >=, \wedge\}$ aplicam-se aos atributos cujos domínios são *valores ordenados*, como os domínios de datas ou numéricos. Os domínios de cadeias (strings) de caracteres são considerados ordenados baseados na sequência dos caracteres. Se o domínio de um atributo é um conjunto de *valores desordenados*, então apenas os operadores de comparação no conjunto $\{=, \wedge\}$ podem ser usados. Um exemplo de um domínio desordenado é o domínio Cor = {vermelho, azul, verde, branco, amarelo,...}, no qual nenhuma ordem é especificada entre as várias cores. Alguns domínios permitem tipos adicionais de operadores de comparação; por exemplo, um domínio de cadeia de caracteres pode permitir o operador de comparação SUBSTRINGJDF.

PNO	MINICIAL	UNOME	SSN	DATANASC	ENDEREÇO	SEXO	SALÁRIO	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston.TX	M	40000	888665555	5
Jennifer		Wallace	987654321	1941-06-20	291 Berry.Bellaire.TX	F	43000	888665555	4
Ramesh		Narayan	666884444	1962-09-15	975 FireOak.Humble.TX	M	38000	333445555	5

UNOME	PNO	SALÁRIO
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

FIGURA 6.1 Resultados das operações SELEÇÃO e PROJEÇÃO, (a) $(T_{(DNO=4 \text{ AND SALARIO}>25000) \text{ OR } (DNO=5 \text{ AND SALARIO}>30000)}(EMPREGADO))$. (b)

$TT_{UNOME \text{ PNO}}$

$SALARIO(EMPREGADO)$. (O $TT_{SEXO \text{ SALARIO}}(EMPREGADO)$).

Em geral, o resultado de uma operação SELEÇÃO pode ser determinado como segue. A <condição de seleção> é aplicada independentemente para cada tupla t em R . Isso é feito substituindo-se cada ocorrência de um atributo A , na condição de seleção, com seu valor na tupla $t[A]$. Se a condição resultar em VERDADEIRO, então a tupla t é **selecionada**. Todas as tuplas selecionadas aparecem no resultado da operação SELEÇÃO. As condições booleanas AND, OR e NOT têm sua interpretação normal, conforme segue:

- (cond1 AND cond2) é VERDADEIRA se ambas, (cond1) e (cond2) , forem VERDADEIRAS; caso contrário, ela será FALSA.
- (cond1 OR cond2) é VERDADEIRA se (cond1) , ou (cond2) , ou ambas forem VERDADEIRAS; caso contrário, ela será FALSA.
- (NOT cond) é VERDADEIRA se cond for FALSA; caso contrário, ela será FALSA.

O operador SELEÇÃO é unário, isto é, ele é aplicado a uma única relação. Além disso, a operação de seleção é aplicada a *cada tupla individualmente*, portanto, as condições de seleção não podem envolver mais que uma tupla. O **grau** da relação resultante de uma operação SELEÇÃO — seu número de atributos — é o mesmo grau de R . O número de tuplas na relação resultante é sempre *menor ou igual* ao número de tuplas em R . Isto é, $I \text{ cr}_c(R) I < I \text{ cr}_c(R)$ para qualquer condição C . A fração de tuplas selecionadas por uma condição de seleção refere-se à **seletividade** da condição.

Observe que a operação SELEÇÃO é **comutativa**, isto é,

$$\sigma_{\text{cond1}}(\sigma_{\text{cond2}}(R)) = \sigma_{\text{cond2}}(\sigma_{\text{cond1}}(R))$$

SEXO	SALÁRIO
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

6.1 Operações Relacionais Unárias: SELEÇÃO (SELECT) e PROJEÇÃO (PROJECT) 109

Portanto, uma seqüência de SELEÇÃO pode ser aplicada em qualquer ordem. Além disso, podemos sempre combinar uma **propagação** de operações SELEÇÃO dentro de uma única operação SELEÇÃO, com uma condição conjuntiva (AND), isto é:

$$CT \langle \text{cond1} \rangle (CT \langle \text{Cond2} \rangle (\dots (CT \langle \text{condn} \rangle (R)) \dots)) = CT \langle \text{cond1} \rangle \text{ AND } \langle \text{cond2} \rangle \text{ AND } \dots \text{ AND } \langle \text{condn} \rangle (R)$$

6.1.2 A Operação PROJEÇÃO

Se pensamos em uma relação como uma tabela, a operação SELEÇÃO seleciona algumas das *linhas* da tabela, enquanto descarta outras. A operação PROJEÇÃO, porém, seleciona certas *colunas* da tabela e descarta outras. Se estivermos interessados apenas em certos atributos de uma relação, usamos a operação PROJEÇÃO para *ressaltá-los*. O resultado da operação PROJEÇÃO pode, portanto, ser visualizado como um *particionamento vertical* da relação em duas relações: uma com as colunas necessárias (atributos) para conter o resultado da operação e a outra com as colunas descartadas. Por exemplo, para listar o primeiro, o último nome e o salário de cada empregado, usamos a operação PROJEÇÃO como segue:

TT UNOME, PNOME, SALARIO (EMPREGADO)

O resultado da relação é mostrado na Figura 6.1 (h). A forma geral da operação PROJEÇÃO é

TT TT<lista de atributo>(R)

em que TT (pi) é o símbolo usado para representar a operação PROJEÇÃO, e <lista de atributo>, a lista dos atributos desejados entre aqueles da relação R. Novamente, observe que R é, em geral, uma *expressão da álgebra relacional* cujo resultado é uma relação que, no caso mais simples, é exatamente o nome de uma relação do banco de dados. O resultado da operação PROJEÇÃO tem apenas os atributos especificados na <lista de atributo>, *na mesma ordem em que eles aparecem na lista*. Portanto, seu **grau** é igual ao número de atributos na <lista de atributo>.

Se a lista de atributos incluir apenas os atributos de R que não forem chave, é provável que ocorram tuplas repetidas. A operação PROJEÇÃO *remove quaisquer tuplas repetidas*, assim, o resultado da operação PROJEÇÃO é um conjunto de tuplas e, portanto, uma relação válida. Isso é conhecido como **eliminação de repetições**. Por exemplo, considere a seguinte operação PROJEÇÃO:

TT SEXO, SALARIO (EMPREGADO)

O resultado é mostrado na Figura 6.1c. Observe que a tupla <F, 25000> aparece apenas uma vez nessa figura, ainda que essa combinação de valores apareça duas vezes na relação EMPREGADO.

O número de tuplas em uma relação resultante de uma operação PROJEÇÃO é sempre menor ou igual ao número de tuplas em R. Se a lista escolhida for uma superchave de R — isto é, incluir alguma das chaves de R —, a relação resultante terá o *mesmo número* de tuplas que R. Além disso,

$$TT \langle \text{lista1} \rangle (TT \langle \text{lista2} \rangle (R)) = TT \langle \text{lista1} \rangle (R)$$

contanto que a <lista2> contenha os atributos em < lista 1 >; do contrário, a expressão do lado esquerdo será incorreta. Também deve ser notado que a comutatividade *não* se aplica em PROJEÇÃO.

6.1.3 Seqüências de Operações e a Operação REBATIZAR (RENAME)

As relações mostradas na Figura 6.1 não têm quaisquer nomes. Em geral, podemos querer aplicar diversas operações de álgebra relacional, uma após a outra. Podemos escrever as operações ou como uma única **expressão de álgebra relacional**, pelo aninhamento das operações, ou podemos aplicar uma operação por vez e criar relações de resultados intermediários. No último caso, devemos dar nomes às relações que envolvem os resultados intermediários. Por exemplo, para recuperar o primeiro nome, o último nome e o salário de todos os empregados que trabalham no departamento número 5, devemos aplicar uma operação SELEÇÃO e uma PROJEÇÃO. Podemos escrever uma única expressão de álgebra relacional como segue:

TT PNOME, UNOME, SALARIO (CT DNO=5(EMPREGADO))

3 Se as repetições não forem eliminadas, o resultado poderia ser um **multiconjunto** (ou *bag*) de tuplas, em vez de um conjunto. Embora isso não seja permitido no modelo formal de relações, ocorre na prática. Veremos, no Capítulo 8, que a SQL permite que o usuário especifique se as tuplas repetidas devem ou não ser eliminadas.

110 Capítulo 6 A Álgebra Relacional e o Cálculo Relacional

A Figura 6.2a mostra o resultado dessa expressão de álgebra relacional. Alternativamente, podemos mostrar explicitamente a sequência de operações, dando um nome para cada relação intermediária:

DEP5_EMPS <- CT_{DNO=5}(EMPREGADO)

RESULTADO <- TT_{PNOOME, UNOME, SALÁRIO}(DEP5_EMPS)

PNOOME	UNOME	SALÁRIO
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

TEMP	PNOOME	MINICIAL	UNOME	SSN	DATANASC	ENDEREÇO	SEXO	SALÁRIO	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren.Houston.TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
	Ramesh	K	Narayan	666884444	1962-09-15	975 FireOak.Humble.TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rico Houston.TX	F	25000	333445555	5

R	PRIMEIRONOME	ULTIMONOME	SALÁRIO
	John	Smith	30000
	Franklin	Wong	40000
	Ramesh	Narayan	38000
	Joyce	English	25000

FIGURA 6.2 Resultados de uma sequência de operações, (a) TT_{PNOOME UNOME SALÁRIO}((T_{DNO=5}(EMPREGADO))). (b) Usando as relações intermediárias e renomeando os atributos.

Freqüentemente, é mais fácil quebrar uma sequência de operações especificando as relações com resultados intermediários que escrever uma única expressão de álgebra relacional. Podemos, também, usar essa técnica para **rebatizar** os atributos nas relações intermediárias e de resultado. Isso pode ser útil na conexão com operações mais complexas, como UNIÃO (UNION) e JUNÇÃO (JOIN), como veremos. Para rebatizar os atributos em uma relação, simplesmente listamos os novos nomes de atributos entre parênteses, como no exemplo a seguir:

TEMP <- CT_{DNO=5}(EMPREGADO)

R(PRIMEIRONOME, ULTIMONOME, SALÁRIO) <- TT_{PNOOME, UNOME, SALÁRIO}(TEMP)

Essas duas operações são ilustradas na Figura 6.2b.

Se nenhum *rename* for aplicado, os nomes dos atributos na relação resultante de uma operação SELEÇÃO serão os mesmos da relação original, e na mesma ordem. Para uma operação PROJEÇÃO sem *rename*, a relação resultante terá os mesmos nomes de atributos que a lista projetada, e na mesma ordem em que eles aparecem na lista.

Podemos também definir uma operação RENAME formal — que pode rebatizar ou o nome da relação, ou os nomes dos atributos, ou ambos — de uma maneira similar àquela que definimos em SELEÇÃO e PROJEÇÃO. A operação geral RENAME, quando aplicada a uma relação R de grau n, é indicada por qualquer uma das três formas seguintes:

PS(B₁, B₂, ..., B_n)(R) OU p_s(R) OU p_(B₁, B₂, ..., B_n)(R)

em que o símbolo P (rho) é usado para indicar o operador RENAME; S, o nome da nova relação; e B₁, B₂, ..., B_n são os novos nomes dos atributos. A primeira expressão rebatiza ambos, a relação e seus atributos; a segunda, apenas a relação; e a terceira, somente os atributos. Se os atributos de R são (A₁, A₂, ..., A_n) nessa ordem, então cada A_i será rebatizado como B_i.

6.2 OPERAÇÕES DA ÁLGEBRA RELACIONAL A PARTIR DA TEORIA DOS CONJUNTOS

6.2.1 As Operações UNIÃO (UNION), INTERSEÇÃO (INTERSECTION) e SUBTRAÇÃO (MINUS)

O próximo grupo de operações de álgebra relacional são as operações matemáticas-padrão em conjuntos. Por exemplo, para recuperar os números do seguro social de todos os empregados que trabalham no departamento 5 ou que supervisionam diretamente um empregado que trabalha no departamento 5, podemos usar a operação UNIÃO conforme segue:

6.2 Operações da Álgebra Relacional a partir da Teoria dos Conjuntos 111

DEP5_EMPS <- CT_{DNO=5}(EMPREGADO) RESULTADO1 <- TT_{SSN}(DEP5_EMPS)
 RESULTADO2(SSN) <- TT_{SUPERSSN}(DEP5_EMPS) RESULTADO <- RESULTADO1 U RESULTADO2

A relação RESULTADO1 tem os números do seguro social de todos os empregados que trabalham no departamento 5, enquanto RESULTADO2 tem os números do seguro social de todos os empregados que supervisionam diretamente um empregado que trabalha no departamento 5. A operação UNIÃO produz as tuplas que estão em RESULTADO1, ou em RESULTADO2, ou em ambos (Figura 6.3). Assim, o valor de SSN 333445555 aparece apenas uma vez no resultado.

Diversos conjuntos de operações teóricas são usados para juntar os elementos de dois conjuntos de várias maneiras, incluindo UNIÃO, INTERSEÇÃO e DIFERENÇA DE CONJUNTO (também chamada SUBTRAÇÃO). Essas são operações binárias, isto é, cada uma é aplicada a dois conjuntos (de tuplas). Quando essas operações são adaptadas a um banco de dados relacional, as duas relações nas quais qualquer uma dessas três operações for aplicada devem ter o mesmo **tipo de tuplas**; essa condição tem sido chamada *compatibilidade de união*. Duas relações $R(A_1, A_2, \dots, A_n)$ e $S(B_1, B_2, \dots, B_n)$ são ditas de **união compatível** se tiverem o mesmo grau n e se o $\text{dom}(A_i)$

$= \text{dom}(B_i)$ para $1 \leq i \leq n$. Isso significa que as duas relações têm o mesmo número de atributos e cada par correspondente de atributos tem o mesmo domínio.

RESULTADO1	SSN
	123456789
	333445555
	666884444
	453453453
RESULTADO2	SSN
	333445555
	888665555
RESULTADO3	SSN
	123456789
	333445555
	666884444
	453453453
	888665555

FIGURA 6.3 Resultado da operação UNIÃO, RESULTADO <- RESULTADO1 U RESULTADO2.

Podemos definir as três operações de UNIÃO, INTERSEÇÃO e DIFERENÇA DE CONJUNTO em duas relações R e S de união compatível, como segue:

- **união**: O resultado dessa operação, indicada por $R \cup S$, é uma relação que inclui todas as tuplas que estão em R , ou em S , ou em ambas, R e S . As tuplas repetidas são eliminadas.
- **interseção**: O resultado dessa operação, indicada por $R \cap S$, é uma relação que inclui todas as tuplas que estão em ambas, R e S .
- **diferença de conjunto** (ou **MINUS** — SUBTRAÇÃO): O resultado dessa operação, indicada por $R - S$, é uma relação que inclui todas as tuplas que estão em R , mas não estão em S .

Adotaremos a convenção de que a relação resultante tem os mesmos nomes de atributos da *primeira* relação R . É sempre possível rebatizar os atributos do resultado usando-se o operador *rename*.

A Figura 6.4 ilustra as três operações. As relações ALUNO e INSTRUTOR na Figura 6.4a são união compatíveis, e suas tuplas representam os nomes dos alunos e instrutores, respectivamente. O resultado da operação UNIÃO na Figura 6.4b mostra os nomes de todos os alunos e instrutores. Observe que tuplas repetidas aparecem apenas uma vez no resultado. O resultado da operação de INTERSEÇÃO (Figura 6.4c) inclui apenas aquelas em ambos, alunos e instrutores.

Observe que tanto UNIÃO quanto INTERSEÇÃO são *operações comutativas*, isto é,

$$R \cup S = S \cup R \quad \text{e} \quad R \cap S = S \cap R$$

Ambas, UNIÃO e INTERSEÇÃO, podem ser tratadas como operações *n-árias* aplicáveis a qualquer número de relações, porque ambas são *operações associativas*, isto é,

$$R \cup (S \cup T) = (R \cup S) \cup T \quad \text{e} \quad (R \cap S) \cap T = R \cap (S \cap T)$$

A operação SUBTRAÇÃO *não é comutativa*; assim, em geral,

$$R - S \neq S - R$$

A Figura 6.4d apresenta os nomes dos alunos que não são instrutores, e a Figura 6.4e mostra os nomes dos instrutores que não são alunos.

112 Capítulo 6 A Álgebra Relacional e o Cálculo Relacional

ALUNO	PN	UN
	Susan	Yao

	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert
INSTRUTOR	PNOME	UNOME
	John	Smith
	Ricardo	Browne
	Susan	Yao
	Francis	Johnson
	Ramesh	Shah
PN	UN	
Susan	Yao	
Ramesh	Shah	
PN	UN	
Susan	Yao	
Ramesh	Shah	
Johnny	Kohler	
Barbara	Jones	
Amy	Ford	
Jimmy	Wang	
Ernest	Gilbert	
John	Smith	
Ricardo	Browne	
Francis	Johnson	
PN	UN	
Johnny	Kohler	
Barbara	Jones	
Amy	Ford	
Jimmy	Wang	
Ernest	Gilbert	
PNOME	UNOME	
John	Smith	
Ricardo	Browne	
Francis	Johnson	

FIGURA 6.4 As operações de conjunto UNIÃO, INTERSEÇÃO e SUBTRAÇÃO, (a) Duas relações união compatíveis, (b) ALUNO U INSTRUTOR, (c) ALUNO $\dot{\sim}$ INSTRUTOR, (d) ALUNO — INSTRUTOR, (e) INSTRUTOR — ALUNO.

6.2.2 A Operação PRODUTO CARTESIANO (CROSS PRODUCT)

A seguir, discutiremos a operação PRODUTO CARTESIANO — também conhecida como CROSS PRODUCT (PRODUTO CRUZADO) ou CROSS JOIN (JUNÇÃO CRUZADA) —, que é indicada por X. Essa também é uma operação binária de conjunto, mas as relações nas quais ela é aplicada *não* têm de ser união compatíveis. Essa operação é usada para combinar as tuplas de duas relações de forma combinatória. Em geral, o resultado de $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$ é uma relação Q, com grau $n + m$ de atributos $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, nessa ordem. A relação resultante Q tem uma tupla para cada combinação de tuplas — uma de R e uma de S. Portanto, se R tiver n_R tuplas (indicada por $|R| = n_R$), e S tiver n_S tuplas, então $R \times S$ terá $n_R * n_S$ tuplas.

A operação aplicada por si só, em geral, não tem sentido. Ela será útil quando for seguida por uma seleção que casar os valores de determinados atributos provindos das relações componentes. Por exemplo, suponha que queiramos montar uma lista com os nomes dos dependentes de cada empregada. Podemos fazer o seguinte:

```
EMPS_FEM <- CT SEXO=F (EMPREGADO) NOMESEMP <- TT PNAME, UNOME, SSN(EMPS_FEM) DEPENDENTES_EMP <- NOMESEMP X DEPENDENTE
DEPENDENTES_REAIS <- CT SSN=ESSN(DEPENDENTES_EMP) RESULTADO <- TT PNAME, UNOME, NOME_DEPENDENTE(DEPENDENTES_REAIS)
```

As relações resultantes dessa sequência de operações são mostradas na Figura 6.5. A relação DEPENDENTES_EMP é o resultado da aplicação da operação PRODUTO CARTESIANO em NOMESEMP, da Figura 6.5, e DEPENDENTE da Figura 5.6. Em DEPENDENTES_EMP, toda tupla de NOMESEMP é combinada com cada tupla de DEPENDENTE, dando um resultado que não tem muito sentido. Gostaríamos de combinar uma tupla de empregada apenas com os dependentes dela — isto é, as tuplas de DEPENDENTE nas quais os valores de SSN

6.3 Operações Relacionais Binárias: JUNCAO (JOIN) e DIVISÃO (DIVISION) 113

se casam com o valor de SSN da tupla de EMPREGADO. A relação DEPENDENTES_REAIS faz isso. A relação DEPENDENTES_EMP é um bom exemplo da correta aplicação da álgebra relacional para obter os resultados que não teriam sentido nenhum de outro modo. Portanto, é responsabilidade do usuário certificar-se de que sejam aplicadas apenas as operações significativas às relações.

O PRODUTO CARTESIANO cria tuplas com a combinação dos atributos de duas relações. Podemos, então, fazer SELEÇÃO apenas das tuplas relacionadas entre as duas relações, especificando uma condição de seleção apropriada, conforme fizemos no exemplo anterior. Em virtude do fato de essa sequência, PRODUTO CARTESIANO seguido por SELEÇÃO, ser bastante usada para identificar e selecionar tuplas relacionadas entre duas relações, uma operação especial chamada JUNCAO (JOIN) foi criada para especificar essa sequência como uma operação única. Discutiremos a operação JUNÇÃO (JOIN) a seguir.

EMPS_FEM	PNAME	MINICIAL	UNOME	SSN	DATANASC	ENDEREÇO	SEXO	SALÁRIO	SUPERSSN	DNO
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle.Spring.TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry.Bellaire.TX	F	43000	888665555	4
	Joyce	A	English	453453453	1972-07-31	5631 Rice.Houston, TX	F	25000	333445555	5

NOMESEMP	PNAME	UNOME	SSN
	Alicia	Zelaya	999887777
	Jennifer	Wallace	987654321
	Joyce	English	453453453

DEPENDENTES_EMP	PNAME	UNOME	SSN	ESSN	NOME^DEPENDENTE	SEXO	DATANASC			
	Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05			
	Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25			
	Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03			
	Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28			
	Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04			
	Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30			
	Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05			
	Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05			
	Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25			
	Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03			
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28			
	Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04			
	Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30			
	Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05			
	Joyce	English	453453453	333445555	Alice	F	1986-04-05			
	Joyce	English	453453453	333445555	Theodore	M	1983-10-25			
	Joyce	English	453453453	333445555	Joy	F	1958-05-03			
	Joyce	English	453453453	987654321	Abner	M	1942-02-28			
	Joyce	English	453453453	123456789	Michael	M	1988-01-04			
	Joyce	English	453453453	123456789	Alice	F	1988-12-30			
	Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05			

DEPENDENTES_REAIS	PNome	UNome	SSN	ESSN	NOME_DEPENDENTE	SEXO	DATANASC	• •
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	• •
RESULTADO	PNome	UNome	NOME_DEPENDENTE					
	Jennifer	Wallace	Abner					

FIGURA 6.5 A operação PRODUTO CARTESIANO (CROSS PRODUCT).

6.3 OPERAÇÕES RELACIONAIS BINÁRIAS: JUNÇÃO OOIN) E DIVISÃO (DIVISION)

6.3.1 A Operação JUNÇÃO

A operação JUNÇÃO, indicada por \bowtie , é usada para combinar as *tuplas relacionadas* em duas relações dentro de uma tupla única. Essa operação é muito importante para qualquer banco de dados relacional com mais de uma relação, porque nos permite

processar os relacionamentos entre as relações. Para ilustrar a JUNÇÃO, suponha que queiramos recuperar o nome do gerente de cada departamento. Para obter o nome do gerente, precisaremos combinar cada tupla departamento com a tupla empregado na qual o valor do SSN case com o valor do GERSSN da tupla departamento. Faremos isso usando a operação JUNÇÃO e, depois, projetando os atributos necessários do resultado, conforme segue:

```
DEPT_GER <- DEPARTAMENTO RESULTADO <- TTn
GERSSN=SSN (DEPT_GER
EMPREGADO
DnOME, UNOME, PnOME
```

A primeira operação é ilustrada na Figura 6.6. Observe que GERSSN é uma chave estrangeira e que a restrição de integridade referencial desempenha o papel de casar as tuplas com a relação EMPREGADO, à qual faz referência.

A operação JUNÇÃO pode ser definida por um PRODUTO CARTESIANO seguido por uma operação SELEÇÃO. Entretanto, a JUNÇÃO é importante porque é usada com muita frequência quando se especifica as consultas a um banco de dados. Considere o exemplo dado anteriormente para ilustrar o PRODUTO CARTESIANO, que incluiu a seguinte sequência de operações:

```
DEPENDENTES_EMP <- NOMESEMPX DEPENDENTE DEPENDENTES_REAIS<-(TSSN=ESSNDEPENDENTES_EMP
```

Essas duas operações podem ser substituídas por uma única operação JUNÇÃO, como segue:

A forma geral de uma operação JUNÇÃO em duas relações $R(A_1, A_2, \dots, A_n)$ e $S(B_1, B_2, \dots, B_m)$ é $R \bowtie S$

<condicao da juncao>

O resultado da JUNÇÃO é uma relação Q com $n + m$ atributos $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, nessa ordem. Q tem uma tupla para cada combinação de tuplas — uma de R e uma de S — quando a combinação satisfizer a condição de junção. Essa é a principal diferença entre PRODUTO CARTESIANO e JUNÇÃO. Na JUNÇÃO, apenas as combinações de tuplas que satisfizerem a condição de junção aparecerão no resultado, enquanto no PRODUTO CARTESIANO, todas as combinações de tuplas serão incluídas no resultado. A condição de junção é especificada pelos atributos das duas relações R e S , e é avaliada para cada combinação de tuplas. Cada combinação de tupla para a qual a condição de junção for avaliada como VERDADEIRA será incluída na relação resultante Q , como uma tupla combinada única.

Uma condição de junção geral é da forma

<condicao> AND <condicao> AND ... AND <condicao>

em que cada condição é da forma $A \theta B$, A é um atributo de R , B é um atributo de S , θ é um operador de comparação $\{=, <, <=, >, >=, *\}$. Uma operação JUNÇÃO com uma condição de junção geral é chamada JUNÇÃO THETA (THETA JOIN). As tuplas cujos atributos de junção são *null* não aparecem no resultado. Nesse sentido, a operação JUNÇÃO não preserva, necessariamente, todas as informações das relações participantes.

DEPT_GER	D _n OME	DNUMERO	GERSSN		P _n OME	MINICIAL	UNOME	SSN	
	Research	5	333445555	. . .	Franklin	T	Wong	333445555	
	Administration	4	987654321	. . .	Jennifer	S	Wallace	987654321	
	Headquarters	1	888665555		James	E	Borg	888665555	

FIGURA 6.6 Resultado da operação JUNÇÃO (JOIN), DEPT_GER «- DEPARTAMENTO
(«-),
.. EMPREGADO.

6.3.2 As Variações de JUNÇÃO: EQUIJUNÇÃO (EQUIJOIN) e JUNÇÃO NATURAL (NATURAL JOIN)

O uso mais comum de JUNÇÃO envolve as condições de junção apenas em comparações de igualdade. Assim, uma JUNÇÃO em que o único operador de comparação usado for $=$ é chamada EQUIJUNÇÃO. Ambos os exemplos que consideramos aqui foram de EQUIJUNÇÃO. Observe que no resultado de uma EQUIJUNÇÃO sempre teremos um ou mais pares de atributos com valores idênticos em cada tupla. Por exemplo, na Figura 6.6, os valores dos atributos GERSSN e SSN são idênticos em cada tupla de DEPT_GER em razão da condição de junção de igualdade especificada para os dois atributos. Como um, de cada par de atributos

4 Novamente, observe que R e S podem ser quaisquer relações que resultem de expressões de álgebra relacional gerais.

6.3 Operações Relacionais Binárias: JUNCAO (JOIN) e DIVISÃO (DIVISION) | 115

com valores idênticos, é supérfluo, uma nova operação, chamada JUNÇÃO NATURAL — indicada por * —, foi criada, livrando o segundo atributo (supérfluo) da condição de EQUIJUNÇÃO. A definição-padrão de JUNÇÃO NATURAL exige que os dois atributos de junção (ou cada par dos atributos de junção) tenham o mesmo nome em ambas as relações. Se esse não for o caso, uma operação de *rename* deve ser aplicada primeiro.

No exemplo seguinte, primeiro rebatizaremos o atributo DNUMERO de DEPARTAMENTO para DNUM — assim ele terá o mesmo nome do atributo DNUM em PROJETO — então aplicaremos a JUNÇÃO NATURAL:

```
DEPT_PROJ <- PROJETO * P(DNOME DNUM GERSSN GERDATA+NCID< DEPARTAMENTO)
```

A mesma consulta pode ser feita em dois passos, criando-se uma tabela intermediária DEPT, como segue:

```
DEPT <- P(DNOME DNUM GERSSN, GERDATAINICIO)^A DEPARTAMENTO DEPT_PROJ <- PROJETO * DEPT
```

O atributo DNUM é chamado **atributo de junção**. A relação resultante está ilustrada na Figura 6.7a. Na relação DEPT_PROJ, cada tupla combina uma tupla de PROJETO com uma tupla de DEPARTAMENTO, para o departamento que controla o projeto, mas *apenas um atributo de junção* é preservado.

Se os atributos nos quais a junção natural for especificada já *têm os mesmos nomes em ambas as relações*, é desnecessário rebatizá-las. Por exemplo, para aplicar uma junção natural nos atributos DNUMERO de DEPARTAMENTO e DEPT_LOCALIZACOES, será suficiente escrever

```
DEPT_LOCS <- DEPARTAMENTO * DEPT_LOCALIZACOES
```

A relação resultante é mostrada na Figura 6.7b, que combina cada departamento com suas localizações e tem uma tupla para cada localização. Em geral, JUNÇÃO NATURAL é executada pela equiparação de *todos* os pares de atributos que tenham o mesmo nome nas duas relações. É preciso haver uma lista de atributos de junção de cada relação, e cada par correspondente precisa ter o mesmo nome.

DEPT_PROJ	PNOME	PNUMERO	PLOCALIZACAO	DNUM	DNOME	GERSSN	GERDATAINICIO
	ProdutoX	1	Bellaire	5	Research	333445555	1988-05-22
	ProdutoY	2	Sugarland	5	Research	333445555	1988-05-22
	ProdutoZ	3	Houston	5	Research	333445555	1988-05-22
	Automação	10	Stafford	4	Administration	987654321	1995-01-01
	Reorganização	20	Houston	1	Headquarters	888665555	1981-06-19
	Novosbenefícios	30	Stafford	4	Administration	987654321	1995-01-01

DEPT_LOCS	DNOME	DNUMERO	GERSSN	GERDATAINICIO	LOCALIZAÇÃO
	Sede Administrativa	1	888665555	1981-06-19	Houston
	Administração	4	987654321	1995-01-01	Stafford
	Pesquisa	5	333445555	1988-05-22	Bellaire
	Pesquisa	5	333445555	1988-05-22	Sugarland
	Pesquisa	5	333445555	1988-05-22	Houston

FIGURA 6.7 Resultados de duas operações JUNÇÃO NATURAL, (a) DEPT_PROJ <- PROJETO * DEPT. (b) DEPTJ.0CS <- DEPARTAMENTO *

DEPTJ.0CALIZACOES.

Uma definição mais geral, *embora não padronizada* para JUNÇÃO NATURAL, é $Q \leftarrow R * (<i_{sta}i>)_i (<u_{sta}2>)_S$

Nesse caso, $<lista1>$ especifica uma lista de i atributos de R e $<lista2>$ caracteriza uma lista de i atributos de S . As listas são usadas para formar comparações de igualdade entre os pares de atributos correspondentes e as condições são todas AND. Apenas a lista correspondente aos atributos da primeira relação R — $<lista1>$ — é mantida no resultado Q .

Observe que se não houver combinação de tuplas que satisfaça a condição de junção, o resultado de uma JUNÇÃO será uma relação vazia com zero tuplas. Em geral, se R tem n_R tuplas e S tem n_S tuplas, o resultado de uma operação JUNÇÃO $R \overset{M}{\underset{cor}{*}} S$ terá entre zero e $n_R * n_S$ tuplas. O tamanho esperado do resultado da junção, dividido pelo número máximo $n_R * n_S$, leva a uma razão chamada **seletividade de junção**, que é uma propriedade de cada condição de junção. Se não houver ne-

5 JUNÇÃO NATURAL é, basicamente, uma EQUIJUNÇÃO, seguida da remoção dos atributos supérfluos.

Capítulo 6 A Álgebra Relacional e o Cálculo Relacional

numa condição de junção, todas as combinações de tuplas são qualificadas e a JUNÇÃO degenera em um PRODUTO CARTESIANO, também chamado CROSS PRODUCT ou CROSS JOIN.

Como podemos ver, a operação JUNÇÃO é usada para combinar os dados de múltiplas relações e as informações relacionadas podem ser apresentadas em uma única tabela. Essas operações também são conhecidas como **inner joins (junções internas)**, para distingui-las de uma variação diferente de junção, chamada *junção externa* (Seção 6.4-3). Observe que, às vezes, uma junção pode ser especificada entre uma relação e ela própria, como ilustraremos na Seção 6.4-2. A operação JUNÇÃO NATURAL ou EQUIJUNÇÃO também pode ser especificada entre as diversas tabelas, levando a uma *junção múltipla (n-way join)*. Por exemplo, considere a seguinte junção tripla:

((PROJETO ^M_{DNUM=DNOME} ^{DEPARTAMENTO}) ^M_{GERSSN=SSN} ^{EMPREGADO})

Essa junção liga cada projeto ao seu departamento de controle e então relaciona o departamento ao seu empregado gerente. O resultado da rede é uma relação consolidada, na qual cada tupla contém a informação projeto-departamento-gerente.

6.3.3 Um Conjunto Completo de Operações de Álgebra Relacional

Tem sido demonstrado que o conjunto de operações da álgebra relacional {<T, TT, U, -, x> é um conjunto **completo**, isto é, quaisquer das outras operações originais de álgebra relacional podem ser expressas como uma *seqüência de operações desse conjunto*. Por exemplo, a operação INTERSEÇÃO pode ser expressa usando-se UNIÃO e SUBTRAÇÃO, como segue:

$R \cap S = (R \cup S) - (R - S)$

Embora, em termos exatos, a INTERSEÇÃO não seja necessária, não é conveniente especificar essa complexa expressão toda vez que desejarmos uma interseção. Como outro exemplo, uma operação JUNÇÃO pode ser estabelecida por meio de um PRODUTO CARTESIANO seguido de uma operação SELEÇÃO, conforme já discutimos:

$R \bowtie S = (R \times S) \sigma_{\langle \text{condicao} \rangle}$

Similarmente, uma JUNÇÃO NATURAL pode ser especificada por um PRODUTO CARTESIANO precedido por RENAME, e seguido pelas operações SELEÇÃO e PROJEÇÃO. Portanto, as diversas operações JUNÇÃO também *não são exatamente necessárias* para o poder expressivo da álgebra relacional. Entretanto, é importante considerá-las como operações distintas, pois são convenientes e comumente utilizadas em aplicações de banco de dados. Outras operações têm sido incluídas na álgebra relacional mais por conveniência que por necessidade. Discutiremos uma delas — a operação DIVISÃO (DIVISION) — na próxima seção.

6.3.4 A Operação DIVISÃO

A operação DIVISÃO, indicada por +, é útil para um tipo especial de consulta que, às vezes, ocorre em aplicações de banco de dados. Um exemplo é "Recuperar os nomes dos empregados que trabalham em *todos* os projetos em que 'John Smith' trabalha". Para expressar essa consulta utilizando-se a operação de DIVISÃO, proceda como segue. Primeiro, recupere a lista dos números dos projetos em que 'John Smith' trabalha, em uma relação intermediária SMITH_NRPS:

SMITH <- $\sigma_{\langle \text{PNOME} = \text{John} \wedge \text{UNOME} = \text{Smith} \rangle}(\text{EMPREGADO})$ SMITH_NRPS <- $\text{TT}_{\text{PNO}}(\text{TRABALHA_EM} \wedge_{\text{SSN_SSN}}^{\text{SMITH}})$

Depois, crie uma relação que inclua uma tupla <PNO, ESSN> toda vez que um empregado, cujo número do seguro social seja ESSN, trabalhe no projeto cujo número seja PNO, na relação intermediária SSN_DNS:

SSN_DNS <- $\text{TT}_{\text{ESSN_PNO}}(\text{TRABALHA_EM})$

Finalmente, aplique a operação DIVISÃO nas duas relações, que dá o número do seguro social do empregado desejado:

SSNS(SSN) <- $\text{SSN_NRPS} \div \text{SSN_DNS}$

As operações anteriores são mostradas na Figura 6.8a.

Em geral, a operação DIVISÃO é aplicada em duas relações $R(Z) \div S(X)$, em que $X \subseteq Z$. Seja $Y = Z - X$ (portanto, $Z = X \cup Y$), isto é, seja Y o conjunto de atributos de R que não são atributos de S . O resultado da DIVISÃO é uma relação $T(Y)$, que inclui uma tupla t , se as tuplas t_R aparecem em R com $t_R[Y] = t$, e com $t_R[X] = t_s$ para toda tupla t_s em S . Isso significa que, para

6.3 Operações Relacionais Binárias: JUNCAO (JOIN) e DIVISÃO (DIVISION) | 117

uma tupla t aparecer no resultado T da DIVISÃO, os valores em t devem aparecer em R em combinação com toda tupla em S . Observe que na formulação da operação DIVISÃO, as tuplas na relação do denominador restringem a relação do numerador, selecionando aquelas tuplas no resultado iguais a todos os valores presentes no denominador. Não é necessário conhecer quais são aqueles valores.

SSNLNRPS	ESSN	NRP
	123456789	1
	123456789	2
	666884444	3
	453453453	1
	453453453	2
	333445555	2
	333445555	3
	333445555	10
	333445555	20
	999887777	30
	999887777	10
	987987987	10
	987987987	30
	987654321	30
	987654321	20
	888665555	20

SMITH_NRPS	NRP
	1
	2

SSNS	SSN
	123456789
	453453453

R	A	B
	a1	b1
	a2	b1
	a3	b1
	a4	b1
	a1	b2
	a3	b2
	a2	b3
	a3	b3
	a4	b3
	a1	b4
	a2	b4
	a3	b4

S	A
	a1
	a2
	a3
T	B
	b1
	b4

FIGURA 6.8 A operação DIVISÃO, (a) Dividindo SSN_NRPS por SMITH_NRPS. (b) $T \leftarrow R \div S$.

A Figura 6.8b ilustra uma operação de DIVISÃO em que $X = \{A\}$, $Y = \{B\}$ e $Z = \{A, B\}$. Observe que as tuplas (valores) b_1 e b_4 aparecem em R em combinação com todas as três tuplas de S ; é por isso que elas surgem na relação resultante T . Todos os outros

valores de B em R que não aparecerem com todas as tuplas de S não serão selecionados: $fc_{>_2}$ não aparece com a_2 ; e $fc_{>_3}$ não aparece com a^{\wedge} .

A operação de DIVISÃO pode ser expressa como uma sequência de operações TT, x, e —, como segue:

$$U \leftarrow ITY(R) \ T2^{ITY((S \times T_1) - R)}$$

A operação de DIVISÃO é definida, por conveniência, para a transação com as consultas que envolvam "quantificação universal" (Seção 6.6.6), ou *todas* as condições. A maioria das implementações de SGBDR, com a SQL como a linguagem primária de consulta, não implementam a divisão diretamente. A SQL tem uma maneira indireta de tratar com o tipo de consulta ilustrada anteriormente (Seção 8.5.4). A Tabela 6.1 mostra as várias operações básicas de álgebra relacional que discutimos.

6.4 OUTRAS OPERAÇÕES RELACIONAIS

Alguns requisitos comuns de um banco de dados — que são necessários em linguagens de consulta comerciais de SGBDRs — não podem ser executados com as operações de álgebra relacional originais, descritas nas seções 6.1 a 6.3. Nesta seção, definiremos as operações adicionais para expressar esses requisitos. Essas operações acentuam o poder expressivo da álgebra relacional original.

6.4.1 Funções Agregadas e Agrupamento

O primeiro tipo de requisito que não pode ser expresso na álgebra relacional básica é para especificar as **funções** matemáticas **agregadas** em coleções de valores do banco de dados. Os exemplos dessas funções incluem a recuperação da média ou total do salário de todos os empregados ou o número total de tuplas de empregados. Essas funções são usadas em consultas de estatística simples, que resumem as informações das tuplas do banco de dados. As funções comuns aplicadas às coleções de valores numéricos incluem SOMA, MÉDIA, MÁXIMO e MÍNIMO. A função CONTAR é usada para contar as tuplas ou valores.

Tabela 6.1 Operações de Álgebra Relacional

Operação	Proposta	Notação
SELEÇÃO (SELECT)		
PROJEÇÃO (PROJECT)		
JUNÇÃO THETA (THETAJOIN) EQUIJUNÇÃO (EQUIJOIN)		
JUNÇÃO NATURAL (NATURAL JOIN)		
UNIÃO (UNION)		
INTERSEÇÃO (INTERSECTION)		
Seleciona todas as tuplas que satisfizerem a condição de SELEÇÃO de uma relação R.		
Produz uma nova relação com apenas alguns dos atributos de R e remove as tuplas repetidas.		
Produz todas as combinações de tuplas de R, e R, que satisfizerem a condição de junção.		
Produz todas as combinações de tuplas de R, e R ₂ que satisfizerem uma condição de junção apenas com as comparações de igualdade.		
Igual à EQUIJUNÇÃO, exceto que os atributos de junção de R ₂ não estarão incluídos na relação resultante; se os atributos de junção tiverem os mesmos nomes, não precisarão ser especificados de nenhuma forma.		
Produz uma relação que inclui todas as tuplas em R, ou R ₂ , ou ambas, R, e R ₂ ; R ₁ e R ₂ devem ter uma união compatível.		
Produz uma relação que inclui todas as tuplas em R, e R ₂ ; R, e R ₂ devem ter uma união compatível.		

cr

TT.

<lista de atributo:

„>W
(R)

/?, ou
<atributo de juncao1

),(< ,
atributo de juncao 2

.)/? ,
R. ou
* <atributo de

R*R,
„>.(.
atributo de juncao 2

>) R₂ ou
R, U/? ,
/?,

DR,

6.4 Outras Operações Relacionais 119

Tabela 6.1 Operações de Álgebra Relacional, {continuação}

Operação	Proposta	Notação
DIFERENÇA (DIFFERENCE)	Produz uma relação que inclui todas as tuplas em R , que não estão em R_2 ; R , e R_2 devem ter uma união compatível.	$R - R_2$
PRODUTO CARTESIANO (CROSS PRODUCT)	Produz uma relação que tem os atributos de R , e R_2 e inclui, como tuplas, todas as possíveis combinações de tuplas de R , e R_2 .	$R \times R_2$
DIVISÃO (DIVISION)	Produz uma relação $R(X)$ que inclui todas as tuplas $t[X]$ em R , em combinação com todas as tuplas de $R^A(Y)$, em que $Z = XU Z$.	$R(Z) \div R_2(Y)$

Outra necessidade freqüente é o agrupamento de tuplas em uma relação, a partir do valor de alguns de seus atributos, e então aplicar uma função de agregação independente para cada grupo. Um exemplo poderia ser o agrupamento de tuplas de empregado por DNO, de modo que cada grupo incluiria as tuplas de empregados que trabalham no mesmo departamento. Poderíamos, então, listar, para cada valor DNO, digamos, o salário médio dos empregados do departamento ou o número de empregados que trabalham naquele departamento.

Podemos definir uma operação para FUNÇÃO AGREGADA usando o símbolo \wedge (pronunciado "script F") para especificar esse tipo de requisito, como segue:

<atributos de agrupamento> O <lista de funcao> (R)

em que <atributos de agrupamento> é uma lista de atributos da relação especificada em R ; e <lista de funcao> é uma lista de pares (<funcao> <atributo>). Em cada par, <funcao> é uma das funções permitidas — como SOMA (SUM), MÉDIA (AVERAGE), MÁXIMO (MAX), MÍNIMO (MIN), CONTAR (COUNT) — e <atributo> é um atributo da relação especificada por R . A relação resultante tem os atributos de agrupamento mais um atributo para cada elemento na lista de função. Por exemplo, para recuperar cada número de departamento, o número de empregados no departamento e a média salarial deles, enquanto os atributos resultantes são rebatizados como indicado a seguir, escrevemos:

$P'(DNO, NUM_DE_EMPREGADOS, MEDIA_SAL) (DNO \ O \ CONTAR_{SSN}, MEDIA_SALARIO \ (EMPREGADO))$

O resultado dessa operação é mostrado na Figura 6.9a.

No exemplo anterior, especificamos uma lista de nomes de atributos — entre parênteses na operação RENAME — para a relação resultante R . Se nenhuma operação *renome* for aplicada, então os atributos da relação resultante que correspondem à lista de função serão, cada um, a concatenação do nome da função com o nome do atributo, na forma <funcao>_<atributo>. Por exemplo, a Figura 6.9b mostra o resultado da seguinte operação:

$DNO \ S \ CONTAR_{SSN}, MEDIA_SALARIO \ (EMPREGADO)$

Se não há especificação de atributos de agrupamento, as funções são aplicadas a *todas as tuplas* na relação, de forma que a relação resultante terá *apenas uma única tupla*. Por exemplo, a Figura 6.9c mostra o resultado da seguinte operação:

$\$ \ CONTAR_{SSN}, MEDIA_SALARIO \ (EMPREGADO)$

É importante notar que, em geral, as repetições *não são eliminadas* quando uma função de agregação for aplicada; dessa forma, a interpretação normal de funções como a SOMA e a MÉDIA será computada. Vale enfatizar que o resultado da aplicação de uma função agregada é uma relação, não um número escalar — mesmo que ele tenha um único valor. Isso faz da álgebra relacional um sistema fechado.

6 Não há concordância para uma notação única de funções agregadas. Em alguns casos é usado um "script A".

7 Observe que essa é uma notação arbitrária que estamos sugerindo. Não há uma notação-padrão.

8 Em SQL, a opção para a eliminação de repetições, antes da aplicação de uma função agregada, está disponível por meio da inclusão da palavra-chave *DISTINCT* (distinto) (Seção 8.4-4).

(a)

R	DNO	NUM_DE_EMPREGADOS	MEDIA_SAL
	5	4	33250
	4	3	31000
	1	1	55000

(b)

DNO	CONTAR_SSN	MEDIA^SALARIO
5	4	33250
4	3	31000
1	1	55000
CONTAR_SSN		MEDIA_SALARIO
8		35125

(c)

FIGURA 6.9 A operação FUNÇÃO AGREGADA. (a) $P_{(DNO, NUM_DE_EMPREGADOS, MEDIA_SAL)}(EMPREGADO)$, (b) $g(EMPREGADO)$, (c) $g(EMPREGADO)$.

, (EMPREGADO).

6.4.2 Operações de Clausura Recursiva

Outro tipo de operação que, em geral, não pode ser especificada na álgebra relacional original básica, é a **clausura recursiva**. Essa operação é aplicada a um **relacionamento recursivo** entre as tuplas de mesmo tipo, como o relacionamento entre um empregado e um supervisor. Esse relacionamento é descrito pela chave estrangeira SUPERSSN da relação EMPREGADO nas figuras 5.5 e 5.6, e relaciona cada tupla empregado (no papel de subordinado) a outra tupla empregado (no papel de supervisor). Um exemplo de uma operação recursiva é recuperar todos os supervisionados de um empregado e em todos os níveis — isto é, todos os empregados e' diretamente supervisionados por e todos os empregados e'' , diretamente supervisionados por empregado e' ; todos os empregados e''' diretamente supervisionados por empregado e'' ; e assim por diante.

Embora seja correto na álgebra relacional especificar todos os empregados supervisionados por um e em *um nível específico*, é difícil especificar todos os supervisionados em todos os níveis. Por exemplo, para especificar os SSNs de todos os empregados e' diretamente supervisionados — em *nível um* — pelo empregado e , cujo nome é 'James Borg' (Figura 5.6), podemos aplicar a seguinte operação:

```
BORG_SSN <- TT
SSN ^ PNAME ^ JAMES AND UNOME = 'Borg'
(EMPREGADO)
SUPERVISÃO (SSN1, SSN2M RESULTAD01 (SSN) <- TT,
• TT,
SSN, SUPERSSN
(EMPREGADO)
(SUPERVISÃO ^).
SSN2 = SSN
BORG_SSN)
```

Para recuperar todos os empregados supervisionados por Borg em nível 2 — isto é, todos os empregados e'' supervisionados por algum empregado e' , que é diretamente supervisionado por Borg — podemos aplicar outra JUNCÃO ao resultado da primeira consulta, como segue:

```
RESULTAD02 (SSN) <- TT ^N1 (SUPERVISÃO
RESULTAD01)
```

Para obter ambos os conjuntos de empregados supervisionados em níveis 1 e 2 por 'James Borg', podemos aplicar a operação UNIÃO aos dois resultados, como segue:

```
RESULTADO <- RESULTAD02 U RESULTAD01
```

Os resultados dessas consultas estão ilustrados na Figura 6.10. Embora seja possível recuperar os empregados em cada nível e então usar a UNIÃO entre eles, não podemos, em geral, especificar uma consulta como "recupere os subordinados de 'James Borg' em todos os níveis" sem utilizar um mecanismo de laço. Uma operação chamada *clausura transitiva* de relações foi proposta para computar a recursividade no relacionamento tão logo ela deva proceder.

9

A SQL3-padrão inclui a sintaxe para a clausura recursiva.

6.4 Outras Operações Relacionais

121

(SSN de Borg é 888665555) (SSN) (SUPERSSN)

SUPERVISÃO	SSN1	SSN2
	123456789	333445555
	333445555	888665555
	999887777	987654321
	987654321	888665555
	666884444	333445555
	453453453	333445555
	987987987	987654321
	888665555	null

RESULTAD01	SSN
	333445555
	987654321

(Supervisionado por Borg)

RESULTAD02	SSN
	123456789
	999887777
	666884444
	453453453
	987987987

(Supervisionado por subordinados de Borg)

RESULTADO	SSN
	123456789
	999887777
	666884444
	453453453
	987987987
	333445555
	987654321

(RESULTAD01 U RESULTAD02)

FIGURA 6.10 Uma consulta recursiva em dois níveis.

6.4.3 Operações de Junção Externa

Discutiremos, agora, algumas extensões da operação de JUNCAO que são necessárias para especificar certos tipos de consultas. As operações de JUNÇÃO descritas anteriormente casam as tuplas que satisfizerem a condição de junção. Por exemplo, para uma operação de JUNÇÃO NATURAL $R * S$, apenas as tuplas de R que correspondem às tuplas em S — e vice-versa — aparecem no resultado. Portanto, as tuplas sem uma tupla *correspondente* (ou *relacionada*) são eliminadas do resultado de JUNÇÃO. As tuplas com valores *null* nos atributos de junção também são eliminadas. Isso equivale à perda de informação quando se espera que o resultado de JUNÇÃO seja usado para gerar um relatório baseado em todas as informações das relações componentes.

Um conjunto de operações, chamado junções externas (*outer joins*), pode ser utilizado quando queremos manter todas as tuplas em R , ou todas aquelas em S , ou todas aquelas em ambas as relações no resultado da JUNÇÃO, independentemente se elas têm ou não tuplas correspondentes na outra relação. Isso satisfaz a necessidade de consultas nas quais as tuplas de duas tabelas são combinadas pelo casamento de linhas correspondentes, mas sem a perda de qualquer tupla por falta de valores casados. As operações de junção que descrevemos anteriormente, na Seção 6.3, na qual apenas as tuplas casadas são mantidas no resultado, são conhecidas como junções internas (*inner joins*).

Por exemplo, suponha que queiramos uma lista dos nomes de todos os empregados e também o nome dos departamentos que eles gerenciam, *se gerenciarem algum departamento*; se não gerenciarem nenhum, podemos, então, indicá-lo com um valor *null*. Podemos aplicar uma operação JUNÇÃO EXTERNA À ESQUERDA (LEFT OUTER JOIN), indicada por \leftarrow , para recuperar o resultado, como segue:

TEMP \leftarrow (EMPREGADO $\overset{M}{\text{SSN}}$ =GERSSN DEPARTAMENTO) RESULTADO \leftarrow TT $\overset{M}{\text{PNAMEI}}$ $\overset{M}{\text{INICIAL}}$, UNOME, DNOME ($\overset{TEMP}{}$)

A operação JUNÇÃO EXTERNA À ESQUERDA mantém toda tupla na *primeira relação*, ou na relação R à *esquerda*, em $R \leftarrow S$; se nenhuma tupla correspondente é encontrada em S , então os atributos de S no resultado de junção serão preenchidos ou "enchidos" com valores *null*. O resultado dessas operações é mostrado na Figura 6.11.

Uma operação similar, JUNÇÃO EXTERNA À DIREITA (RIGHT OUTER JOIN), indicada por $\overset{M}{\rightarrow}$, mantém todas as tuplas na *segunda relação*, ou relação S à direita, no resultado de $R \overset{M}{\rightarrow} S$. Uma terceira operação, JUNÇÃO EXTERNA TOTAL (FULL OUTER JOIN), indicada por $\overset{M}{\rightarrow}^E$, mantém todas as tuplas em ambas as relações, esquerda e direita, quando não são encontradas

67

as tuplas com os valores casados, preenchendo-as com valores *null*, conforme a necessidade. As três operações de junção externa são parte da SQL2-padrão (Capítulo 8).

RESULTADO	PNOME	MINICIAL	UNOME	DNOME
	John	B	Smith	<i>null</i>
	Franklin	T	Wong	Research
	Alicia	J	Zelaya	<i>null</i>
	Jennifer	S	Wallace	Administration
	Ramesh	K	Narayan	<i>null</i>
	Joyce	A	English	<i>null</i>
	Ahmad	V	Jabbar	<i>null</i>
	James	E	Borg	Headquarters

FIGURA 6.11 O resultado de uma OPERAÇÃO LEFT OUTER JOIN.

6.4.4 A Operação UNIÃO EXTERNA

A operação UNIÃO EXTERNA (OUTER UNION) foi desenvolvida tomando-se a união de tuplas de duas relações, se as relações *não forem união compatíveis*. Essa operação tomará a UNIÃO de tuplas em duas relações $R(X, Y)$ e $S(X, Z)$, que são **parcialmente compatíveis**, significando que apenas alguns dos seus atributos, digamos X , são de união compatível. Os atributos que são de união compatível são representados apenas uma vez no resultado, e aqueles atributos que não são de união compatível, advindos de qualquer uma das relações, também serão mantidos na relação resultado $T(X, Y, Z)$.

Duas tuplas, t_1 em R e t_2 em S , são ditas casadas se $t_1 \uparrow X = t_2 \uparrow X$, e considera-se que representam a mesma instância de entidade ou relacionamento. Elas serão combinadas (unidas) em uma tupla única T . As tuplas em qualquer uma das relações que não têm tuplas casadas na outra relação serão preenchidas com valores *null*. Por exemplo, uma UNIÃO EXTERNA pode ser aplicada às duas relações cujos esquemas são ALUNO (Nome, SSN, Departamento, Orientador) e INSTRUTOR (Nome, SSN, Departamento, Grau). As tuplas das duas relações serão casadas quando tiverem a mesma combinação de valores de atributos compartilhados — Nome, SSN, Departamento. A relação resultado, ALUNO_JOINSTRUTOR, terá os seguintes atributos:

ALUNO_JOINSTRUTOR (Nome, SSN, Departamento, Orientador, Grau)

Todas as tuplas de ambas as relações serão incluídas no resultado, mas as tuplas com a mesma combinação (Nome, SSN, Departamento) aparecerão apenas uma vez no resultado. As tuplas que surgirem apenas em ALUNO terão um *null* para o atributo Grau, enquanto as tuplas que aparecerem apenas em INSTRUTOR terão um *null* para o atributo Orientador. Uma tupla que existir em ambas as relações, como um aluno que for também um instrutor, terá valores para todos os seus atributos.

Observe que a mesma pessoa pode, ainda, aparecer duas vezes no resultado. Por exemplo, poderíamos ter um aluno, graduado no departamento de Matemática, que é um instrutor no departamento de Ciência da Computação. Embora as duas tuplas representando essa pessoa em ALUNO e INSTRUTOR tenham os mesmos valores (Nome, SSN), elas não concordarão quanto ao valor do Departamento e, assim, não serão casadas. Isso porque Departamento tem dois significados separados em ALUNO (o departamento em que a pessoa estuda) e INSTRUTOR (o departamento em que a pessoa está empregada como um instrutor). Se quiséssemos unir as pessoas com base apenas na mesma combinação (Nome, SSN), deveríamos rebatizar o atributo Departamento em cada tabela, para mostrar que eles têm significados diferentes, e não indicá-los como parte dos atributos de união compatível.

Outra capacidade que existe na maioria das linguagens comerciais (mas não na álgebra relacional básica) é a especificação de operações nos valores, depois que eles foram extraídos do banco de dados. Por exemplo, operações aritméticas, como $+$, $-$ e $*$, podem ser aplicadas a valores numéricos que aparecem no resultado de uma consulta.

6.5 EXEMPLOS DE CONSULTAS (QUERIES) NA ÁLGEBRA RELACIONAL

Daremos, agora, exemplos adicionais para ilustrar o uso das operações de álgebra relacional. Todos os exemplos se referem ao banco de dados da Figura 5.6. Em geral, a mesma consulta pode existir de várias maneiras, usando-se as várias operações. Expressaremos cada consulta de uma maneira e deixaremos que o leitor sugira as formulações equivalentes.

10 Observe que a UNIÃO EXTERNA é equivalente a uma JUNÇÃO EXTERNA TOTAL, se os atributos de junção forem todos os atributos comuns às duas relações.

6.5 Exemplos de Consultas (Queries) na Álgebra Relacional 123

CONSULTA 1

Recupere o nome e endereço de todos os empregados que trabalham para o departamento de 'Pesquisa'.

```
DEPT_PESQUISA <- <Junt< 'PESQUISA', (DEPARTAMENTO) EMP_PESQUISA <- (DEPT_PESQUISA * DNUMERO=DNOEMPREGADO) RESULTADO <- TT_pNOME wmf
ENDERECO(PESQUISA.EMPS)
```

Essa consulta poderia ser especificada de outras formas; por exemplo, a ordem das operações JUNÇÃO e SELEÇÃO poderia ser ao contrário, ou a JUNÇÃO poderia ser substituída por uma JUNÇÃO NATURAL, depois de rebatizar (*rename*) um dos atributos de junção.

CONSULTA 2

Para todo projeto localizado em 'Stafford', liste o número do projeto, o número do departamento de controle e o último nome, endereço e data de nascimento do gerente do departamento.

```
STAFFORD.PROJS <- O_pLOCALIZACAO=>STAFFORD(PROJETO)
CONTR_DEPT <- (STAFFORD.PROJS * "%(DEPARTAMENTO)
PROJ_DEPT_GER <- (CONTR_DEPT ^ GERSN_SSN EMPREGADO)
RESULTADO <- TT_pNUMERO DNUM UNOME ENDERECO, DATANASC( PROJ_DEPT_MGR)
```

CONSULTA 3

Encontre os nomes dos empregados que trabalham em todos os projetos controlados pelo departamento número 5.

```
DEPT5_PROJS(PNO) *^TT_pNUMERO(DNUM=5 (PROJETO)) EMP_PROJ(SSN, PNO) <- TT_ESSN_pNO (TRABALHA_EM) RESULTADO_EMP_SSNS <- EMP_PROJ +
DEPT5_PROJS RESULTADO <- TT_UNOME_pNOME(RESULTADO_EMP_SSNS * EMPREGADO)
```

CONSULTA 4

Faça uma lista dos números dos projetos que envolvam um empregado cujo último nome seja 'Smith', mesmo que esse trabalhador seja o gerente do departamento que controla o projeto.

```
SMITHS(ESSN) <- TT_SSN((X_T_UNOME=-SMITH- (EMPREGADO))
PROJ_EMPREGADO_SMITH <- TT_pNO(TRABALHA_EM * SMITHS)
GERS <- TT_UNOME DNUMERO(EMPREGADO "SSN=GERSN" DEPARTAMENTO)
DEPTS_GERENCIADO_SMITH(DNUM) <- ifDNUMERO(CTUNCIME=-SMITH-(GERS)) PROJ_GER_SMITH(PNO) <-
TT_NUMERO(DEPTS_GERENCIADO_SMITH * PROJETO) RESULTADOS- (PROJ_EMPREGADO_SMITH U DEPTS_GERENCIADO_SMITH)
```

CONSULTA 5

Liste os nomes de todos os empregados com dois ou mais dependentes.

Em termos exatos, essa consulta não poderá ser feita na *álgebra relacional básica (original)*. Temos de usar a operação FUNÇÃO AGREGADA com a função agregada CONTAR. Vamos supor que os dependentes do *mesmo* empregado tenham valores para NOME_DEPENDENTE *distintos*.

```
n(SSN, NUM_DE_DEPTS) <- ESSN ^ CONTAR_NOME_DEPENDENTE (DEPENDENTE)
" <- "NUM_DEPTS>2 (' * '
RESULTADO <- TT_UNOME_pNOME (// 'EMPREGADO)
```

CONSULTA 6

Recupere os nomes de empregados que não tenham dependentes.

Capítulo 6 A Álgebra Relacional e o Cálculo Relacional

Este é um exemplo do tipo de consulta que usa a operação SUBTRAÇÃO (MINUS — DIFERENÇA DE CONJUNTOS).

```
TODOS_EMPES <- TT_SSN (EMPREGADO) EMPES_COM_DEPS(SSN) <- TT_SSN (DEPENDENTE) EMPES_SEM_DEPS <- (TODOS_EMPES - EMPES_COM_DEPS)
RESULTADO <- TT_UNOME PNOME (EMPES_COM_DEPS * EMPREGADO)
```

CONSULTA 7

Liste os nomes dos gerentes que tenham, pelo menos, um dependente.

```
GERES(SSN) <- TT_GERSSN (DEPARTAMENTO) EMPES_COM_DEPS(SSN) <- TT_SSN (DEPENDENTE) GERES_SEM_DEPS <- (GERES CI EMPES_COM_DEPS)
RESULTADO <- TT_UNOME PNOME (GERES_COM_DEPS * EMPREGADO)
```

Conforme mencionamos anteriormente, a mesma consulta pode, em geral, ser especificada de muitas maneiras diferentes. Por exemplo, as operações podem, freqüentemente, ser aplicadas em várias ordens. Além disso, algumas operações podem ser usadas para substituir outras; por exemplo, a operação INTERSEÇÃO, na Consulta 7, pode ser substituída por uma JUNÇÃO NATURAL. Como exercício, tente fazer cada uma das consultas dos exemplos anteriores utilizando as operações diferentes. No Capítulo 8 e nas seções 6.6 e 6.7, mostraremos como essas consultas são escritas em outras linguagens relacionais.

6.6 O CÁLCULO RELACIONAL DE TUPLA

Nesta e na próxima seção, introduziremos outra linguagem formal de consulta para o modelo relacional, chamada **cálculo relacional**. No cálculo relacional, escrevemos uma expressão **declarativa** para expressar um requisito de recuperação, portanto, não será feita nenhuma descrição de como uma consulta se desenvolve. Uma expressão de cálculo especifica *o que* será recuperado, em vez de *como* recuperá-lo. Desse modo, o cálculo relacional é considerado uma linguagem **não procedural**. Difere da álgebra relacional, em que precisamos escrever uma *seqüência de operações* para especificar um requisito de recuperação, assim ela pode ser considerada um caminho **procedural** para determinar uma consulta. É possível aninhar as operações de álgebra para formar uma expressão única, entretanto, a ordem determinada entre as operações sempre é explicitamente especificada na expressão de álgebra relacional. Essa ordem também influencia a estratégia para a evolução da consulta. Uma expressão de cálculo pode ser escrita de diferentes maneiras, mas a forma em que ela é escrita não corresponde a como ela será desenvolvida.

Tem sido demonstrado que qualquer recuperação que puder ser especificada na álgebra relacional básica também poderá ser especificada no cálculo relacional e vice-versa; em outras palavras, o **poder** expressivo das duas linguagens é *idêntico*. Isso leva à definição do conceito de uma linguagem **relacionalmente completa**. Uma linguagem relacional de consulta L é considerada **relacionalmente completa** se pudermos expressar em L qualquer consulta que possa ser expressa em cálculo relacional. A 'completeza' relacional tem se tornado uma base comparativa importante para o poder de expressividade de linguagens de consulta de alto nível. Entretanto, como dissemos na Seção 6.4, certas consultas requisitadas freqüentemente em aplicações de banco de dados não podem ser expressas na álgebra ou cálculo relacional básico. Em sua maioria, as linguagens de consulta relacional são relacionalmente completas, mas têm *maior poder expressivo* que a álgebra relacional ou o cálculo relacional, por causa das operações adicionais, como as funções agregadas, agrupamentos e ordenações.

Nesta e na próxima seção, todos os nossos exemplos novamente se referem ao banco de dados mostrado nas figuras 5.6 e 5.7. Usaremos as mesmas consultas que foram utilizadas na Seção 6.5. As seções 6.6.5 e 6.6.6 discutem o tratamento de quantificadores universais e podem ser omitidas por alunos interessados em uma introdução geral ao cálculo de tupla.

11 Quando as consultas são otimizadas (Capítulo 15), o sistema escolherá uma seqüência particular de operações que correspondam a uma estratégia de execução eficiente.

6.6.1 Variáveis de Tuplas e Relações-Limite

O cálculo relacional de tupla está baseado na especificação de um número de variáveis de tupla. Cada variável de tupla geralmente *abrange* uma particular relação do banco de dados, significando que a variável pode tomar, como seu valor, qualquer tupla individual daquela relação. Uma simples consulta de cálculo de tupla relacional é da forma

$\{t \mid \text{COND}(t)\}$

em que t é uma variável de tupla e $\text{COND}(t)$, uma expressão condicional envolvendo t . O resultado dessa consulta é o conjunto de todas as tuplas t que satisfizerem a $\text{COND}(t)$. Por exemplo, para encontrar todos os empregados cujo salário está acima de 50 mil dólares, podemos escrever a seguinte expressão de cálculo de tupla:

$\{t \mid \text{EMPREGADO}(t) \text{ AND } t.\text{SALARIO} > 50000\}$

A condição $\text{EMPREGADO}(t)$ especifica que a relação-limite da variável de tupla t é o EMPREGADO. Cada tupla EMPREGADO t que satisfizer a condição $t.\text{SALARIO} > 50000$ será recuperada. Observe que $t.\text{SALARIO}$ faz referência ao atributo SALÁRIO da variável de tupla t ; essa notação compõe os nomes dos atributos com os nomes das relações ou seus apelidos em SQL, como veremos no Capítulo 8. Na notação do Capítulo 5, $t.\text{SALARIO}$ é o mesmo que escrever $\bar{t}[\text{SALARIO}]$.

A consulta anterior recupera todos os valores de atributo para cada tupla t de EMPREGADO selecionada. Para recuperar apenas *alguns* dos atributos — digamos, os primeiros e os últimos nomes —, escrevemos

$\{t.\text{PNOME}, t.\text{UNOME} \mid \text{EMPREGADO}(t) \text{ AND } t.\text{SALARIO} > 50000\}$

Informalmente, precisamos especificar a seguinte informação em uma expressão de cálculo de tupla:

- Para cada variável de tupla t , a relação-limite R de t . Esse valor é especificado por uma condição da forma $R(t)$.
- Uma condição para selecionar as combinações particulares de tuplas. As variáveis de tupla abrangem suas respectivas relações-limite, a condição é avaliada para toda combinação possível de tuplas de modo a identificar as combinações selecionadas para aquelas cujas condição for avaliada como VERDADEIRA.
- Um conjunto de atributos para ser recuperado, os atributos requisitados. Os valores desses atributos são recuperados para cada combinação de tuplas selecionada.

Antes de discutirmos a sintaxe formal de cálculo relacional de tupla, considere outra consulta.

CONSULTA 0

Recupere a data de nascimento e o endereço do empregado (ou empregados) cujo nome seja 'John B. Smith'.

QO: $\{t.\text{DATANASC}, t.\text{ENDEREÇO} \mid \text{EMPREGADO}(t) \text{ AND } t.\text{PNOME} = \text{'John'} \text{ AND } t.\text{MINIT} = \text{'B'} \text{ AND } t.\text{UNOME} = \text{'Smith'}\}$

No cálculo relacional de tuplas, especificamos primeiro os atributos requisitados $t.\text{DATANASC}$ e $t.\text{ENDEREÇO}$ para cada tupla t selecionada. Depois, estabelecemos a condição para selecionar uma tupla após a barra (\mid) — a saber, seja t uma tupla da relação EMPREGADO cujos valores dos atributos PNOME, MINICIO e UNOME são 'John', 'B' e 'Smith', respectivamente.

6.6.2 Expressões e Fórmulas em Cálculo Relacional de Tupla

Uma expressão geral de cálculo relacional de tupla é da forma

$\{t_1.A_1, t_2.A_2, \dots, t_n.A_n \mid \text{COND}(t_1, t_2, \dots, t_n, t_{n+1}, t_{n+2}, \dots, t_{n+m})\}$

em que $t_1, t_2, \dots, t_n, t_{n+1}, \dots, t_{n+m}$ são variáveis de tuplas, cada A_i é um atributo da relação à qual t_i está limitado e COND refere-se a uma condição ou fórmula¹² do cálculo relacional de tupla. Uma fórmula é feita com os átomos de cálculo de predicado, que podem ser um dos seguintes:

1. Um átomo da forma $R(t)$, em que R é um nome de relação e t uma variável de tupla. Esse átomo identifica os limites da variável de tupla t , como a relação cujo nome seja R .
2. Um átomo da forma $t_1.A \text{ op } t_2.B$, em que op é um dos operadores de comparação no conjunto $\{=, <, <=, >, >=, *\}$, t_1 e t_2 são variáveis de tuplas, A é um atributo da relação na qual t_1 está limitado, e B é um atributo da relação na qual t_2 está limitada.

¹² Também chamada fórmula bem definida (*uiell-fortned formula*) ou *wff*, na lógica matemática.

126 Capítulo 6 A Álgebra Relacional e o Cálculo Relacional

3. Um átomo da forma $t_i.A \text{ op } c \text{ ou } c \text{ op } t_i.B$, em que op é um dos operadores de comparação no conjunto $\{=, <, <=, >, *\}$; t_j e t_i são variáveis de tupla; A corresponde a um atributo da relação na qual t_i está limitado; B refere-se a um atributo da relação na qual t_i está limitada; e c , a um valor constante.

Cada um dos átomos anteriores evoluem para VERDADEIRO ou FALSO para uma combinação específica de tuplas; este é chamado **valor verdade** de um átomo. Em geral, uma variável de tupla t abrange todas as possíveis tuplas 'no universo'. Para os átomos da forma $R(t)$, se t for designado para uma tupla que é um *membro da relação R especificada*, o átomo é VERDADEIRO; caso contrário, ele será FALSO. Nos átomos de tipos 2 e 3, se as variáveis de tuplas forem designadas para as tuplas de forma que os valores dos atributos especificados nas tuplas satisfaçam a condição, então o átomo será VERDADEIRO.

Uma **fórmula** (condição) é feita de um ou mais átomos conectados pelos operadores lógicos **AND (E)**, **OR(OU)** e **NOT (NÃO)**, e é definida recursivamente como segue:

1. Todo átomo é uma fórmula.
2. Se F_1 e F_2 são fórmulas, então assim são $(F_1 \text{ AND } F_2)$, $(F_1 \text{ OR } F_2)$, **NOT(F_1)** e **NOT(F_2)**. Os valores verdade dessas fórmulas são derivados de suas fórmulas componentes F_1 e F_2 , como segue:
 - a. $(F_1 \text{ AND } F_2)$ será VERDADEIRO se ambos, F_1 e F_2 , forem VERDADEIROS; caso contrário, será FALSO.
 - b. $(F_1 \text{ OR } F_2)$ será FALSO se ambos, F_1 e F_2 , forem FALSOS; caso contrário, será VERDADEIRO.
 - c. **NOT(F_1)** será VERDADEIRO se F_1 for FALSO; será FALSO se F_1 for VERDADEIRO.
 - d. **NOT(F_2)** será VERDADEIRO se F_2 for FALSO; será FALSO se F_2 for VERDADEIRO.

6.6.3 Os Quantificadores Universais e Existenciais

Além disso, dois símbolos especiais, chamados **quantificadores**, podem aparecer nas fórmulas; eles são o **quantificador universal (V)** e o **quantificador existencial (3)**. Os valores verdade para as fórmulas com quantificadores são descritos nas regras 3 e 4 a seguir; embora primeiro precisemos definir os conceitos de variáveis de tupla livre e vinculada em uma fórmula. Informalmente, uma variável de tupla t será vinculada se ela for quantificada, significando que aparece em uma cláusula $(3 t)$ ou (Vt) ; caso contrário, ela será livre. Formalmente, definiremos uma variável de tupla em uma fórmula como **livre** ou **vinculada** de acordo com as seguintes regras:

- Uma ocorrência de uma variável de tupla em uma fórmula F que é um átomo é livre em F .
- Uma ocorrência de uma variável de tupla t é livre ou vinculada em uma fórmula feita de conectivos lógicos — $(F_1 \text{ AND } F_2)$, $(F_1 \text{ OR } F_2)$, **NOT(F_1)** e **NOT(F_2)** —, dependendo se ela for livre ou vinculada em F_1 ou F_2 (se ocorrer em uma das duas). Observe que em uma fórmula na forma $F = (F_1 \text{ AND } F_2)$ ou $F = (F_1 \text{ OR } F_2)$, uma variável de tupla pode ser livre em F_1 e vinculada em F_2 ou vice-versa; nesse caso, uma ocorrência da variável de tupla será vinculada e a outra, livre em F .
- Todas as ocorrências livres de uma variável de tupla t em F serão **vinculadas** em uma fórmula F' da forma $F' = (3 t)(F)$ ou $F' = (Vt)(F)$. A variável de tupla é vinculada para o quantificador especificado em F' . Por exemplo, considere as seguintes fórmulas:
 $O.DNOME='PESQUISA'$
 $(3T)(D.DNUMERO=T.DN0)$
 $(Vd)(D.GERSSN)=333445555'$

A variável de tupla d é livre em ambas, F_1 e F_2 , enquanto é vinculada para o quantificador (V) em F_3 . A variável T é vinculada para o quantificador (3) em F_2 .

Daremos, agora, as regras 3 e 4 para a definição de uma fórmula que iniciamos anteriormente:

3. Se F é uma fórmula, então também o será $(3 t)(F)$, em que t é uma variável de tupla. A fórmula $(3 t)(F)$ será VERDADEIRA se a fórmula F evoluir para VERDADEIRO em *algumas* (pelo menos uma) tuplas designadas como ocorrências livres de t em F ; caso contrário, $(3 t)(F)$ será FALSA.
4. Se F é uma fórmula, então também o será $(V t)(F)$, em que t é uma variável de tupla. A fórmula $(V t)(F)$ será VERDADEIRA se a fórmula F evoluir para VERDADEIRO em *todas as tuplas* (no universo) designadas como ocorrências livres de t em F ; caso contrário, $(Vt)(F)$ será FALSA.

6.6 O Cálculo Relacional de Tupla

127

O quantificador (3) é conhecido como um quantificador existencial porque uma fórmula (3 t)(F) será VERDADEIRA se existir alguma tupla que faça F VERDADEIRA. Para o quantificador universal, (Vt)(F) será VERDADEIRA se toda tupla que puder ser designada para as ocorrências livres de t em F for substituída por t, e F for VERDADEIRA para *cada uma dessas substituições*. É chamado quantificador universal ou 'para todos' porque toda tupla no 'universo de' tuplas deve fazer F VERDADEIRA para tomar a fórmula quantificada VERDADEIRA.

6.6.4 Consultas de Exemplo Usando o Quantificador Existencial

Usaremos algumas das mesmas consultas da Seção 6.5 para dar uma idéia de como as mesmas consultas são especificadas na álgebra relacional e no cálculo relacional. Observe que algumas consultas são especificadas mais facilmente na álgebra relacional que no cálculo relacional e vice-versa.

CONSULTA 1

Recupere o nome e o endereço de todos os empregados que trabalham para o departamento de 'Pesquisa'.

Q1: {t.PNOME, t.UNOME, t. ENDEREÇO | EMPREGADO(t) AND 3 (d) (DEPARTAMENTO(d) AND d.DNOME = 'Pesquisa' AND d.DNUMERO = t.DNO) }

As *únicas variáveis de tupla livres* em uma expressão de cálculo relacional deveriam ser aquelas que aparecem do lado esquerdo da barra (|). Em Q1, t é a única variável livre; depois ela será *sucessivamente vinculada* a cada tupla. Se uma tupla *satisfizer as condições* especificadas em Q1, os atributos PNOME, UNOME e ENDEREÇO serão recuperados para cada uma das tuplas. As condições EMPREGADO(t) e DEPARTAMENTO(d) especificam as relações-limite para t e d. A condição d.DNOME = 'Pesquisa' é uma condição de seleção e corresponde a uma operação SELEÇÃO na álgebra relacional, enquanto a condição d.DNUMERO = t.DNO é uma **condição de junção** e serve a uma proposta similar à da operação JUNÇÃO (Seção 6.3).

CONSULTA 2

Para cada projeto localizado em 'Stafford', liste o número do projeto, o número do departamento de controle e o último nome, data de nascimento e endereço do gerente de departamento.

Q2: {p.PNUMERO, p.DNUM, m.UNOME, m.DATANASC, m.ENDEREÇO | PROJETO(p) AND EMPREGADO(m) AND p.PLOCALIZACAO = 'Stafford' AND ((3d)(DEPARTAMENTO(d) AND p.DNUM=d.DNUMERO AND d.GERSSN=m.SSN)) }

Em Q2 há duas variáveis de tupla livres, p e m. A variável de tupla d é vinculada ao quantificador existencial. A condição de consulta é avaliada para toda combinação de tuplas designadas para p e m; e tirando todas as possíveis combinações de tuplas cujos p e m forem vinculadas, apenas as combinações que satisfizerem a condição são selecionadas.

Diversas variáveis de tupla em uma consulta podem abranger a mesma relação. Por exemplo, para especificar a consulta Q8 — para cada empregado, recupere o primeiro e o último nome do empregado, e o último nome do seu supervisor imediato —, especificamos duas variáveis de tupla e e s, em que ambas abrangem a relação EMPREGADO:

Q8: {e.PNOME, e.UNOME, s.PNOME, s.UNOME | EMPREGADO(e) AND EMPREGADO(s) AND e.SUPERSSN=s.SSN}

CONSULTA 3

Encontre o nome de cada empregado que trabalhe em algum projeto controlado pelo departamento número 5. Essa é uma variação da consulta 3, na qual 'todo' é trocado por 'algum'. Neste caso, precisaremos de duas condições de junção e de dois quantificadores existenciais.

Q3': {e.UNOME, e.PNOME | EMPREGADO(e) AND ((3 x) (3 w) (PROJETO(x) AND TRABALHA_EM(w) AND x.DNUM=5 AND w.ESSN=e.SSN AND x.PNUMERO=w.PNO)) }

CONSULTA 4

Faça uma lista dos números dos projetos que envolvam um empregado cujo último nome seja 'Smith', como empregado ou gerente do departamento que controle o projeto.

Q4: {p.PNUMERO | PROJETO(p) AND ((3 e) (3 w)(EMPREGADO(e) AND TRABALHA_EM(w) AND w.PNO=p.PNUMERO AND e.UNOME='Smith' AND e.SSN=w.ESSN)) }

128

Capítulo 6 A Álgebra Relacional e o Cálculo Relacional

ou

$(\exists m)(\exists d)(\text{EMPREGADO}(m) \text{ AND } \text{DEPARTAMENTO}(cf) \text{ AND } f.d.NUM=f.d.NUMERO \text{ AND } d.GERSSN=m.ssN \text{ AND } m.UNOME='Smith'))$

Compare essa com a versão em álgebra relacional dessa consulta, na Seção 6.5. A operação UNIÃO na álgebra relacional pode, geralmente, ser substituída por um conectivo *OR* (OU) no cálculo relacional. Na próxima seção discutiremos o relacionamento entre os quantificadores universal e existencial, e mostraremos como um pode ser transformado no outro.

6.6.5 Transformando os Quantificadores Universal e Existencial

Introduziremos, agora, algumas transformações bem estabelecidas na lógica matemática que relacionam os quantificadores universal e existencial. É possível transformar um quantificador universal em um quantificador existencial e vice-versa, para conseguir uma expressão equivalente. Uma transformação geral pode ser descrita informalmente como segue: transformar um tipo de quantificador em outro com negativa (precedida por NOT - NÃO); AND (E) e OR (OU) substituem um ao outro; uma fórmula de negação torna-se não-negação; e uma fórmula não-negação torna-se de negação. Alguns casos especiais dessa transformação podem ser iniciados como segue, em que o símbolo = significa equivalente a:

$(\forall x) (P(x)) \text{ J NOT } (\exists x) (\text{NOT } (P(x)))$
 $(\exists x) (P(x)) \text{ J NOT } (\forall x) (\text{NOT } (P(x)))$
 $(\forall x) (P(x) \text{ AND } Q(x)) \text{ J NOT } (\exists x) (\text{NOT } (P(x)) \text{ OR NOT } (Q(x)))$
 $(\forall x) (P(x) \text{ OR } Q(x)) \text{ j NOT } (\exists x) (\text{NOT } (P(x)) \text{ AND NOT } (Q(x)))$
 $(\exists x) (P(x) \text{ OR } Q(x)) \text{ J NOT } (\forall x) (\text{NOT } (P(x)) \text{ AND NOT } (Q(x)))$
 $(\exists x) (P(x) \text{ AND } Q(x)) \text{ j NOT } (\forall x) (\text{NOT } (P(x)) \text{ OR NOT } (Q(x)))$

Observe, também, que o seguinte é VERDADEIRO, em que o símbolo \Rightarrow significa implica:

$(\forall x)(P(x)) \Rightarrow (\exists x)(P(x))$

$\text{NOT } (\exists x) (P(x)) \Rightarrow \text{NOT } (\forall x) (P(x))$

6.6.6 Usando o Quantificador Universal

Se usarmos um quantificador universal, é inteiramente judicioso seguir umas poucas regras para garantir que nossa expressão tenha sentido. Discutiremos essas regras com respeito à Consulta 3.

CONSULTA 3

Encontre os nomes dos empregados que trabalhem em *todos* os projetos controlados pelo departamento número 5. Um meio para especificar essa consulta é usar o quantificador universal, como mostrado.

Q3: $\{e.UNOME, e.PNOME \mid \text{EMPREGADO}(e) \text{ AND } (\forall x) (\text{NOT}(\text{PROJETO}(x) \text{ OR NOT } (x.DNUM=5) \text{ OU } ((\exists w) (\text{TRABALHA_EM}(w) \text{ AND } w.ESSN=e.ssN \text{ AND } x.PNUMERO=w.PNO))))\}$

Podemos quebrar Q3 em seus componentes básicos, como segue:

Q3: $\{e.UNOME, e.PNOME \mid \text{EMPREGADO}(e) \text{ AND } F\}$

$F' = ((\forall x) (\text{NOT}(\text{PROJETO}(x) \text{ OR } F_i))$

$F_i = \text{NOT } (x.DNUM=5) \text{ OR } F_2$

$F_2 = ((\exists w) (\text{TRABALHA_EM}(w) \text{ AND } w.ESSN=e.ssN \text{ AND } x.PNUMERO=w.PNO))$

Queremos ter a certeza de que um empregado selecionado e trabalhe em *todos os projetos* controlados pelo departamento 5, mas a definição de quantificador universal diz que, para tornar a fórmula quantificada VERDADEIRA, a fórmula interna deverá ser VERDADEIRA *para todas as tuplas do universo*. O truque é excluir da quantificação universal todas as tuplas que não nos interessarem, fazendo com que a condição torne-se VERDADEIRA *para todas essas tuplas*. Isso é necessário porque uma variável de tupla quantificada universalmente, como é x em Q3, precisa evoluir para VERDADEIRO *para toda possível tupla* designada a ela que torne a fórmula quantificada VERDADEIRA. As primeiras tuplas excluídas (fazendo com que evoluam automaticamente para VERDADEIRO) são aquelas que não estão na relação R de interesse. Em Q3, usando a expressão $\text{NOT}(\text{PROJETO}(x))$ dentro da fórmula quantificada universalmente, evoluirão para VERDADEIRO todas as tuplas x que não

6.6 O Cálculo Relacional de Tupla

129

estiverem na relação PROJETO. Assim, excluiríamos as tuplas de R que não nos interessam. Em Q3, usando a expressão $\text{NOT}(x.\text{DNUM}=5)$, evoluiremos para VERDADEIRO todas as tuplas x que estejam na relação PROJETO, mas que não sejam controladas pelo departamento 5. Finalmente, especificamos uma condição F_2 que precisa valer para todas as tuplas que permanecerem em R . Portanto, podemos explicar Q3 como segue:

1. Para a fórmula $F' = (\forall x) (F)$ ser VERDADEIRA, precisamos que a fórmula F seja VERDADEIRA *para todas as tuplas no universo que possam ser designadas para x* . Entretanto, em Q3 estamos interessados apenas em F sendo VERDADEIRA para todas as tuplas da relação PROJETO que sejam controladas pelo departamento 5. Portanto, a fórmula F é da forma $(\text{NOT}(\text{PROJETO}(x)) \text{ OR } F_i)$. A condição ' $\text{NOT}(\text{PROJETO}(x)) \text{ OR } \dots$ ' será VERDADEIRA para todas as tuplas *que não estiverem na relação PROJETO*, e tem o efeito de eliminar aquelas tuplas da avaliação do valor verdade de F_j . Para toda tupla na relação PROJETO, F_i deverá ser VERDADEIRA se F for VERDADEIRA.

2. Usando a mesma linha de raciocínio, não queremos considerar tuplas na relação PROJETO que não sejam controladas pelo departamento número 5, uma vez que estamos interessados apenas nas tuplas PROJETO cujo $\text{DNUM} = 5$. Portanto, podemos escrever:

SE $(x.\text{DNUM}=5)$ ENTÃO F_2

o que é equivalente a

$(\text{NOT}(x.\text{DNUM}=5) \text{ OR } F_2)$

3. A fórmula F , portanto, é da forma $\text{NOT}(x.\text{DNUM}=5) \text{ OR } F_2$. No contexto de Q3, isso significa que, para uma tupla x na relação PROJETO, ou seu DNUM for 5 ou deverá satisfazer F_2 .

4. Finalmente, F_2 estabelece a condição para controlar uma tupla EMPREGADO selecionada que queremos: que o empregado trabalhe no PROJETO *de toda tupla que ainda não tiver sido excluída*. Essas tuplas empregado serão selecionadas pela consulta.

Em português, Q3 dá a seguinte condição para selecionar uma tupla EMPREGADO e : para toda tupla x na relação PROJETO, com $x.\text{DNUM}=5$, deve existir uma tupla w em TRABALHA_EM tal que $w.\text{ESSN} = e.\text{SSN}$ e $w.\text{PNO} = x.\text{PNUMERO}$. Isso equivale a dizer que EMPREGADO e trabalha em todo PROJETO x no DEPARTAMENTO número 5 (Ufa!).

Usando a transformação geral dos quantificadores universal para existencial, dada na Seção 6.6.5, podemos rephrasear a consulta em Q3 como mostrado em Q3A:

Q3A: $\{ \text{CUNOME}, \hat{e}.\text{PNOME} \mid \text{EMPREGADO}(e) \text{ AND } (\text{NOT } (\exists x) (\text{PROJETO}(x) \text{ AND } (x.\text{DNUM}=5) \text{ AND } (\text{NOT } (\exists w) (\text{TRABALHA_EM}(w) \text{ AND } w.\text{ESSN}=e.\text{SSN} \text{ AND } x.\text{PNUMERO}=w.\text{PNO})))) \}$

Agora, damos alguns exemplos adicionais de consultas que usam os quantificadores.

CONSULTA 6

Encontre os nomes dos empregados que não tenham dependentes.

Q6: $\{ \text{CPNOME}, \hat{e}.\text{UNOME} \mid \text{EMPREGADO}(\hat{e}) \text{ AND } (\text{NOT } (\exists d) (\text{DEPENDENTE}(d) \text{ AND } e.\text{ssN}=d.\text{ESSN})) \}$

Usando a regra de transformação geral, podemos rephrasear Q6 como segue:

Q6A: $\{ e.\text{PNOME}, \text{CUNOME} \mid \text{EMPREGADO}(\hat{e}) \text{ AND } ((\forall d) (\text{NOT}(\text{DEPENDENTE}(d)) \text{ OR } \text{NOT } (e.\text{ssN}=d.\text{ESSN}))) \}$

CONSULTA 7

Liste os nomes dos gerentes que tenham, pelo menos, um dependente. Q7: $\{ e.\text{PNOME}, \hat{e}.\text{UNOME} \mid \text{EMPREGADO}(e) \text{ AND } ((\exists d) (\exists p) (\text{DEPARTAMENTO}(d) \text{ AND } \text{DEPENDENTE}(p) \text{ AND } e.\text{SSN}=\hat{e}.\text{GERSSN} \text{ AND } p.\text{ESSN}=e.\text{ssN})) \}$

Essa consulta, interpretada como 'gerentes que tenham, pelo menos, um dependente' passa para 'gerentes para quais exista algum dependente'.

89

6.6.7 Expressões Seguras

Quando usamos os quantificadores universais, os quantificadores existenciais ou negação de predicados em uma expressão de cálculo, devemos ter certeza de que a expressão resultante fará sentido. Uma **expressão segura** em cálculo relacional é aquela que garante que seja produzido um *número finito de tuplas* no resultado; caso contrário, a expressão é **arriscada**, como é conhecida.

Por exemplo, a expressão

$(t \mid \text{NOT}(\text{EMPREGADO}(t)))$

é **arriscada** porque produz todas as tuplas no universo que *não* sejam tuplas EMPREGADO, que é infinitamente numeroso. Se seguirmos as regras anteriormente discutidas para Q3, obteremos uma expressão segura quando usarmos os quantificadores universais. Podemos definir as expressões seguras, mais precisamente, introduzindo o conceito de *expressão de cálculo relacional de domínio de uma tupla*: que é o conjunto de todos os valores que aparecem como valores constantes na expressão ou que existam em alguma tupla nas relações referidas pela expressão. O domínio de $\{t \mid \text{NOT}(\text{EMPREGADO}(t))\}$ é o conjunto de todos os valores de atributos que apareçam em alguma tupla da relação EMPREGADO (para qualquer atributo). O domínio da expressão Q3A deveria incluir todos os valores que apareçam em EMPREGADO, PROJETO e TRABALHA_EM (unidos com o valor 5 aparecendo ele mesmo na consulta).

Uma expressão é dita **segura** se todos os valores em seu resultado são do domínio da expressão. Observe que o resultado de $(t \mid \text{NOT}(\text{EMPREGADO}(t)))$ é arriscado, uma vez que, em geral, ele incluirá tuplas (portanto, os valores) de fora da relação EMPREGADO; esses valores não estão no domínio da expressão. Todos os nossos outros exemplos são considerados expressões seguras.

6.7 O CÁLCULO RELACIONAL DE DOMÍNIO

Há outro tipo de cálculo relacional chamado cálculo relacional de domínio ou, simplesmente, **cálculo de domínio**. Enquanto a SQL (Capítulo 8), uma linguagem baseada no cálculo relacional de tupla, estava sendo desenvolvida pela IBM Research em San José, Califórnia, outra linguagem, chamada QBE (*Query-By-Example* — Consulta-Por-Exemplo), que está relacionada ao cálculo de domínio, estava sendo desenvolvida quase simultaneamente pela IBM Research em Yorktown Heights, Nova York. A especificação formal de cálculo de domínio foi proposta depois do desenvolvimento do sistema QBE.

Cálculo de domínio difere do cálculo de tupla pelo tipo de *variáveis* usadas nas fórmulas: em vez de ter variáveis abrangendo as tuplas, as variáveis abrangem os valores únicos dos domínios dos atributos. Para formar uma relação de grau n , o resultado de uma consulta, devemos ter n dessas **variáveis de domínio** — uma para cada atributo. Uma expressão de cálculo de domínio é da forma

$\{X_1, X_2, \dots, X_n \mid \text{COND}(X_1, X_2, \dots, X_n, X_{n+1}, X_{n+2}, \dots, X_{n+m})\}$

em que $X_1, X_2, \dots, X_n, X_{n+1}, X_{n+2}, \dots, X_{n+m}$ são variáveis de domínio que abrangem os domínios (de atributos), e COND é uma **condição** ou **fórmula** do cálculo relacional de domínio.

Uma fórmula é feita de **átomos**. Os átomos de uma fórmula são ligeiramente diferentes daqueles do cálculo de tuplas e podem ser um dos seguintes:

1. Um átomo da forma $R(x_1, x_2, \dots, x_i)$, no qual R é o nome de uma relação de grau j e cada x_i , $1 \leq i \leq j$, uma variável de domínio. Esse átomo significa que uma lista de valores de $\langle x_1, x_2, \dots, x_j \rangle$ deve ser uma tupla na relação cujo nome é R , em que x_i é o valor do i -ésimo valor de atributo da tupla. Para fazer uma expressão de cálculo de domínio mais concisa, podemos *tirar as vírgulas* da lista de variáveis; assim, podemos escrever

$\{x_1, x_2, \dots, x_n \mid R(x_1, x_2, x_3) \text{ AND } \dots\}$

em vez de

$\{x_1, x_2, \dots, x_n \mid R(x_1, x_2, x_3) \text{ AND } \dots\}$

2. Um átomo da forma $x_i \text{ op } x_j$, em que op é um dos operadores de comparação do conjunto $\{=, <, >, \geq, \leq, *\}$, e x_i e x_j são as variáveis de domínio.

3. Um átomo da forma $x_i \text{ op } c$ ou $c \text{ op } x_i$, no qual op é um dos operadores de comparação do conjunto $\{=, <, >, \geq, \leq, *\}$, x_i e c são as variáveis de domínio, e c é um valor constante.

6.7 O Cálculo Relacional de Domínio

131

Como no cálculo de tupla, os átomos evoluem para VERDADEIRO ou para FALSO em um dado conjunto de valores, chamado **valores verdade** dos átomos. No caso 1, se as variáveis de domínio forem valores designados correspondentes aos da tupla de uma dada relação R, então o átomo será VERDADEIRO. Nos casos 2 e 3, se as variáveis de domínio forem valores designados que satisfaçam a condição, então o átomo é VERDADEIRO.

De forma similar ao cálculo relacional de tupla, as fórmulas são construídas a partir de átomos, variáveis e quantificado-res, assim, não repetiremos as especificações para as fórmulas aqui. Seguem alguns exemplos de consultas especificadas em cálculo de domínio. Usaremos as letras minúsculas l, m, n, \dots, x, y, z para as variáveis de domínio.

CONSULTA 0

Recupere a data de nascimento e o endereço do empregado cujo nome seja 'John B. Smith'.

QO: $\{uv \mid I(3q)(3r)(3s)(3t)(3w)(3x)(3y)(3z) \mid (iMPKGADo(qrstuvwxyz) \text{ AND } <2='JOHN' \text{ AND } r='B' \text{ AND } s='SMITH')\}$

Precisamos de dez variáveis para a relação EMPREGADO, uma para abranger cada domínio de atributo, na ordem. Das dez variáveis q, r, s, \dots, z , apenas uev são livres. Especificaremos, primeiro, *os atributos solicitados*, DATANASC e ENDEREÇO, pelas variáveis de

domínio livres u para DATANASC e v para ENDEREÇO. Depois, especificaremos a condição para selecionar uma tupla, seguindo a barra (|) — assim, se a sequência de valores designados às variáveis $qrstuvwxyz$ for uma tupla da relação EMPREGADO, os valores para q (PNOME), r (MINICIAL) e s (UNOME) seriam 'John', 'B' e 'Smith', respectivamente. Por conveniência, quantificaremos apenas aquelas variáveis que *aparecem, de fato, em uma condição* (essas poderiam ser q, r e s em QO), no restante de nossos exemplos.

Uma notação alternativa taquigráfica, usada em QBE, para escrever essa consulta, seria designar as constantes 'John', 'B' e 'Smith' diretamente como mostrado em QOA. Aqui, todas as variáveis que não aparecem à esquerda da barra são, implicitamente, quantificadas por existência.

QOA: $\{uv \mid I(EMPREGAD0('John', 'B', 'Smith', t, u, v, w, x, y, z))\}$

CONSULTA 1

Recupere o nome e o endereço de todos os empregados que trabalhem para o departamento de 'Pesquisa'.

Q1: $\{qsv \mid I(3z)(3i)(3m) \mid (EMPREGADo(qrstuvwxyz) \text{ AND } DEPARTAMENTO(imno) \text{ AND } !='PESQUISA' \text{ AND } m=z)\}$

Uma condição relacionando duas variáveis de domínio que abranjam os atributos de duas relações, como $m = z$ em Q1, é uma **condição de junção**; enquanto uma condição que relaciona uma variável de domínio a uma constante, como $I = 'Pesquisa'$, é uma **condição de seleção**.

CONSULTA 2

Para todo o projeto localizado em 'Stafford', liste o número do projeto, o número do departamento de controle e o último nome, data de nascimento e endereço do gerente de departamento.

Q2: $\{iksv \mid I(3j)(3m)(3n)(3t) \mid (PROJETO(hijlc) \text{ AND } EMPREGADo(qrstuiwry?) \text{ AND } DEPARTAMENTO(imno) \text{ AND } k=m \text{ AND } n=t \text{ AND } J='STAFFORD')\}$

CONSULTA 6

Encontre os nomes de empregados que não tenham dependentes.

Q6: $\{qs \mid I(3t) \mid (EMPREGADo(<jrstuviwQ>5:) \text{ AND } (NOT(3i) \mid (DEPENDENTE(imnop) \text{ AND } t=i)))\}$

A Consulta 6 pode ser redefinida usando-se os quantificadores universais em vez de quantificadores existenciais, como mostrado em Q6A:

Q6A: $\{qs \mid I(3t) \mid (EMPREGADo(qrstuvw^?:) \text{ AND } ((\forall i) \mid (NOT(DEPENDENTE(lmnop)) \text{ OR } NOT(t=i))))\}$

13 Observe que a notação que quantifica somente as variáveis de domínio realmente utilizadas nas condições, e mostrar um predicado como EMPREGADO(qrstwuoo/:j) sem separar as variáveis de domínio com vírgulas, é uma notação abreviada usada por conveniência; não é a notação formal correta.

14 Novamente, essa não é uma notação formal exata.

CONSULTA 7

Liste os nomes dos gerentes que tenham, pelo menos, um dependente.

Q7: $\{sq\ I(t)/j(l)\ (iMPREGADo(qrstuvwxyz)\ AND\ DEPARTAMENTo(hijk)\ AND\ DEPENDENTE(Imnop)\ AND\ t=j\ AND\ i=t)\}$

Conforme mencionamos anteriormente, pode ser mostrado que qualquer consulta, que pode ser expressa na álgebra relacional, também pode ser expressa no domínio ou cálculo relacional de tupla. Também, qualquer *expressão de segurança* no domínio ou cálculo de tupla relacional pode ser expressa na álgebra relacional.

A linguagem *Query-By-Example* (QBE — Consulta-Por-Exemplo) foi baseada no cálculo relacional de domínio; embora este tenha sido compreendido mais tarde, depois o cálculo de domínio foi formalizado. A QBE foi uma das primeiras linguagens gráficas de consulta, com sintaxe mínima desenvolvida para os sistemas de banco de dados. Foi desenvolvida pela IBM Research e está disponível como um produto comercial da empresa, como parte da opção de interface QMF (*Query Management Facility* — Facilidade de Gerenciamento de Consulta) para DB2. Vem sendo imitada por diversos produtos comerciais. Por causa do seu importante emprego no campo das linguagens relacionais, incluímos uma visão de QBE no Apêndice D.

6.8 RESUMO

Neste capítulo apresentamos duas linguagens formais para o modelo de dados relacional. Elas são usadas para manipular as relações e produzir novas relações como respostas para as consultas. Discutimos a álgebra relacional e suas operações, que são empregadas para especificar uma seqüência de operações de uma consulta. Depois, introduzimos dois tipos de cálculo relacional, conhecidos como cálculo de tupla e cálculo de domínio; eles são declarativos à medida que especificam o resultado de uma consulta, sem determinar como produzir o resultado da consulta.

Nas seções 6.1 a 6.3 introduzimos as operações básicas de álgebra relacional e ilustramos os tipos de consultas para cada uma. Os operadores relacionais unários SELEÇÃO e PROJEÇÃO, bem como a operação RENAME, foram discutidos primeiro. Depois, discutimos as operações binárias teóricas de conjunto, que exigem que as relações às quais serão aplicadas sejam de união compatível; estas incluem UNIÃO, INTERSEÇÃO e DIFERENÇA DE CONJUNTO. A operação PRODUTO CARTESIANO é uma operação de conjunto que pode ser usada para combinar as tuplas de duas relações, produzindo todas as suas possíveis combinações. Raramente é usado na prática, entretanto, mostramos como o PRODUTO CARTESIANO, seguido por uma SELEÇÃO, pode ser usado para definir os casamentos de tuplas entre duas relações e levar a uma operação de JUNÇÃO. Diferentes operações de JUNÇÃO, chamadas JUNÇÃO THETA, EQUIJUNÇÃO e JUNÇÃO NATURAL foram introduzidas.

Discutimos alguns tipos importantes de consultas que *não podem* ser declaradas a partir das operações básicas da álgebra relacional, mas que são importantes para as situações práticas. Introduzimos a operação de FUNÇÃO AGREGADA para lidar com os tipos agregados de solicitações. Discutimos as consultas recursivas, para as quais não há suporte direto na álgebra, mas que podem ser abordadas passo a passo, como demonstramos. Depois, apresentamos as operações de JUNÇÃO EXTERNA e UNIÃO EXTERNA, que são extensões de JUNÇÃO e UNIÃO, e que permitem que todas as informações nas relações-fonte sejam preservadas no resultado.

As duas últimas seções descreveram os conceitos básicos sob o cálculo relacional, que é baseado em um ramo da lógica matemática chamada cálculo de predicado. Há dois tipos de cálculo relacional: (1) o cálculo relacional de tupla, que usa as variáveis de tupla que abrangem as tuplas (linhas) das relações e (2) o cálculo relacional de domínio, que usa as variáveis de domínio que englobam os domínios (colunas das relações). No cálculo relacional, uma consulta é especificada em uma afirmação declarativa única, sem especificar qualquer ordem ou método para recuperar o resultado da consulta. Portanto, o cálculo relacional é freqüentemente considerado uma linguagem de alto nível, em vez da álgebra relacional, pois uma expressão de cálculo relacional declara *o que* queremos recuperar, independentemente de *como* a consulta deva ser executada.

Discutimos a sintaxe de consultas de cálculo relacional usando variáveis de tupla e domínio. Discutimos, também, o quantificador existencial (\exists) e o quantificador universal (\forall). Vimos que as variáveis de cálculo relacional são vinculadas a esses quantificadores. Descrevemos em detalhes como são escritas as consultas com a quantificação universal, e debatemos o problema de especificação de consultas seguras, cujos resultados são finitos. Discutimos, também, as regras para transformação de quantificadores universais em existenciais e vice-versa. São os quantificadores que dão o poder de expressão ao cálculo relacional, fazendo-o equivalente à álgebra relacional. Não há analogia para as funções de agregação e agrupamento no cálculo básico relacional, embora algumas extensões tenham sido sugeridas.

6.8 Resumo 133

Questões de Revisão

- 6.1. Liste as operações da álgebra relacional e a proposta de cada uma.
- 6.2. O que é compatibilidade de união? Por que as operações UNIÃO, INTERSEÇÃO e DIFERENÇA exigem que as relações nas quais elas forem aplicadas sejam união compatíveis?
- 6.3. Discuta alguns tipos de consultas para as quais o 'rebatismo' (*rename*) dos atributos seja necessário, como forma de evitar ambigüidade na formulação da consulta.
- 6.4. Discuta os vários tipos de operações de *junção interna*. Por que a junção theta é necessária?
- 6.5. Qual o papel que o conceito de *chave estrangeira* desempenha quando se especificam os tipos mais comuns de operações de junção significativas?
- 6.6. O que é a operação FUNÇÃO? Para que é usada?
- 6.7. Como as operações de JUNÇÃO EXTERNA diferem das operações de JUNÇÃO INTERNA? Como as operações de UNIÃO EXTERNA diferem de UNIÃO?
- 6.8. Em que sentido o cálculo relacional difere da álgebra relacional e em que sentido eles são similares?
- 6.9. Como o cálculo relacional de tupla difere do cálculo relacional de domínio?
- 6.10. Discuta o significado do quantificador existencial (\exists) e do quantificador universal (\forall).
- 6.11. Defina os seguintes termos em relação ao cálculo de tupla: *variável de tupla*, *relação-limite*, *átomo*, *fórmula* e *expressão*.
- 6.12. Defina os seguintes termos em relação ao cálculo de domínio: *variável de domínio*, *relação-limite*, *átomo*, *fórmula* e *expressão*.
- 6.13. O que se entende por uma *expressão segura* no cálculo relacional?
- 6.14. Quando uma linguagem de consulta é dita relacionalmente completa?

Exercícios

- 6.15. Mostre os resultados de cada um dos exemplos de consultas da Seção 6.5, aplicadas ao estado do banco de dados da Figura 5.6.
- 6.16. Especifique as seguintes consultas no esquema de banco de dados da Figura 5.5 usando os operadores relacionais discutidos neste capítulo. Mostre também o resultado de cada consulta se ela fosse aplicada ao estado de banco de dados da Figura 5.6.
 - a. Recupere os nomes de todos os empregados do departamento 5 que trabalhem mais de dez horas por semana no projeto 'ProdutoX'.
 - b. Liste os nomes de todos os empregados que tenham um dependente com o mesmo primeiro nome que o deles.
 - c. Encontre os nomes de todos os empregados que são diretamente supervisionados por 'Franklin Wong'.
 - d. Para cada projeto, liste o nome do projeto e o total de horas por semana (de todos os empregados) gastas no projeto.
 - e. Recupere os nomes de todos os empregados que trabalhem em todos os projetos.
 - f. Recupere os nomes de todos os empregados que não trabalham em nenhum projeto.
 - g. Para cada departamento, recupere o nome do departamento e a média salarial de todos os empregados que trabalhem nesse departamento.
 - h. Recupere a média salarial de todos os empregados do sexo feminino.
 - i. Encontre os nomes e os endereços de todos os empregados que trabalhem em pelo menos um projeto localizado em Houston, mas cujo departamento não se localiza em Houston.
 - j. Liste os últimos nomes de todos os gerentes de departamento que não tenham dependentes.
- 6.17. Considere o esquema do banco de dados COMPANHIA AÉREA, mostrado na Figura 5.8, que foi descrito no Exercício 5.11. Especifique as seguintes consultas na álgebra relacional:
 - a. Para cada voo, liste o número do voo, o aeroporto de chegada para o primeiro trecho de voo e o aeroporto de chegada para o último trecho de voo.
 - b. Liste os números dos voos e os dias da semana de todos os voos ou trechos de voo que partam do Aeroporto Internacional de Houston (código do aeroporto 'IAH') e cheguem ao Aeroporto Internacional de Los Angeles (código do aeroporto 'LAX').
 - c. Liste o número do voo, o código do aeroporto de partida, o horário programado para a partida, o código do aeroporto de chegada, o horário programado para a chegada e os dias da semana de todos os voos ou trechos de voo que partam de algum dos aeroportos da cidade de Houston e cheguem em algum dos aeroportos da cidade de Los Angeles.
 - d. Liste todas as informações dos passageiros do voo de número 'C0197'.

Capítulo 6 A Álgebra Relacional e o Cálculo Relacional

e. Recupere o número de poltronas disponíveis para o voo de número 'C0197' em '1999-10-09'.

6.18. Considere o esquema de banco de dados relacional BIBLIOTECA, mostrado na Figura 6.12, que é usado para manter o controle de livros, usuários e livros emprestados. As restrições de integridade referencial são mostradas como arcs direcionados, na Figura 6.12, como na notação da Figura 5.7. Escreva as expressões relacionais para as seguintes consultas:

- Quantas cópias do livro de título *The lost tribe* pertencem à filial cujo nome é 'Sharpstown'?
- Quantas cópias do livro *The lost tribe* pertencem a cada filial da biblioteca?
- Recupere os nomes de todos os usuários que não tenham nenhum registro de empréstimo.
- Para cada livro emprestado na filial de 'Sharpstown' e cuja data de devolução seja hoje, recupere o título do livro, o nome e o endereço do usuário.
- Para cada filial da biblioteca, recupere seu nome e o número total de livros emprestados por ela.
- Recupere os nomes, endereços e número de livros emprestados de todos os usuários que tenham emprestado mais de cinco livros.
- Para cada livro cujo autor (ou co-autor) seja 'Stephen King', recupere o título e o número de cópias possuídas pela filial da biblioteca cujo nome seja 'Central'.

6.19. Especifique as seguintes consultas em álgebra relacional no esquema de banco de dados do Exercício 5.13:

- Liste o Pedido# e Pdata para todos os pedidos remetidos pelo Depósito de número 'W2'.
- Liste as informações do Depósito do qual o Cliente chamado 'José Lopez' teve seus pedidos fornecidos. Produza uma listagem: Pedido#, Depósito#.
- Produza uma listagem NOMECLI, #DOSPEDIDOS, TOTAL_MEDIA_PEDIDO, em que a coluna do meio é o número total de pedidos feitos pelo cliente, e a última coluna é a média dos valores totais dos pedidos desse cliente.
- Liste os pedidos que não foram remetidos até 30 dias da data do pedido.
- Liste o Pedido# dos pedidos que foram remetidos de todos os depósitos cuja a empresa seja em Nova York.

6.20. Especifique as seguintes consultas em álgebra relacional no esquema de banco de dados dado no Exercício 5.14:

- Dê os detalhes (todos os atributos da relação VIAGEM) para as viagens que excederem dois mil dólares em despesas.
- Imprima o SSN do vendedor que viajou para 'Honolulu'.
- Imprima o valor total das despesas de viagem contraídas pelo vendedor com SSN = '234-56-7890'.

LIVRO

AUTORESJJVRO

IdLivro	Titulo	NomeEditora	
IdFilial	NomeFilial	Endereço	
NoCartao	Nome	Endereço	Fone

FIGURA 6.12 Um esquema de banco de dados relacional para um banco de dados BIBLIOTECA.

6.21. Especifique as seguintes consultas em álgebra relacional no esquema de banco de dados dado no Exercício 5.15:

- Liste o número de cursos feitos por todos os alunos chamados 'John Smith' no inverno de 1999 (isto é, Trimestre = 'W99').
- Produza uma lista dos livros-texto (incluir Curso#, ISBN_Livro, Titulo_Livro) para os cursos oferecidos pelo departamento 'CS' que tenham usado mais de dois livros.
- Liste qualquer departamento cujos livros adotados tenham sido todos publicados pela 'AWL Publishing'.

6.22. Considere as duas tabelas T1 e T2 mostradas na Figura 6.13. Mostre os resultados das seguintes operações: a- $T1 \cap T2$ =

6.8 Resumo 135

t>- TI.MTL.Q-T2.B.T2

c. TI.MTL.P = T2.AT2

d. TI.M; TI.Q - T2.B T2

e. TIUT2

f. TI M(TI.p = T2.AANDTI.R = T2.C) T2

6.23. Especifique as seguintes consultas em álgebra relacional no esquema de banco de dados do Exercício 5.16:

a. Para a vendedora chamada 'Jane Doe', liste as seguintes informações para todos os carros vendidos por ela: Série#, Fabricante, Preço_Venda.

b. Liste Série# e Modelo de carros que não tenham opcionais.

c. Considere a operação JUNÇÃO NATURAL entre VENDEDOR e VENDAS. Qual é o significado de uma JUNÇÃO EXTERNA à esquerda para essas tabelas (não mude a ordem das relações). Explique com um exemplo.

d. Escreva uma consulta em álgebra relacional envolvendo a seleção e uma operação de conjunto e expresse, em palavras, o que a consulta faz.

6.24. Especifique as consultas a, b, c, e, f, i e j do Exercício 6.16 em ambos os cálculos relacionais, de tupla e domínio.

6.25. Especifique as consultas a, b, c e d do Exercício 6.17 em ambos os cálculos relacionais, de tupla e domínio.

6.26. Especifique as consultas c, d, f e g do Exercício 6.18 em ambos os cálculos relacionais, de tupla e domínio.

Tabela T1

A	B	C
P	Q	R

Tabela T2

10	a	5	10	b	6
15	b	8	25	c	3
25	c	6	10	b	5

FIGURA 6.13 Um estado do banco de dados para as relações T1 e T2.

6.27. Em uma consulta de cálculo relacional de tupla, com n variáveis de tupla, qual seria o número mínimo de condições de junção? Por quê? Qual é o efeito de ter um número menor de condições de junção?

6.28. Reescreva as consultas de cálculo relacional de domínio que seguiram Q0 na Seção 6.7, no estilo de notação abreviada de QOA, na qual o objetivo é minimizar o número de variáveis de domínio escrevendo as constantes no lugar de variáveis, sempre que possível.

6.29. Considere essa consulta: recupere os SSNs dos empregados que trabalhem, pelo menos, naqueles projetos nos quais o empregado com SSN = 123456789 trabalhe. Ela deve ser declarada como (PARATODO x) (SE P ENTÃO Q), em que

- x é uma variável de tupla que abrange a relação PROJETO.
- P = empregado com SSN = 123456789 trabalha no projeto x.
- Q = empregado e trabalha no projeto x.

Expresse a consulta em cálculo relacional de tupla usando as regras

- $(\forall x) (P(x)) = \text{NOT}(\exists x) (\text{NOT}(P(x)))$.
- $(\text{SE } P \text{ ENTÃO } Q) = (\text{NOT}(P) \text{ OR } Q)$.

6.30. Mostre como você deve especificar as seguintes operações de álgebra relacional em ambos os cálculos relacionais, de tupla e domínio.

a. $CT_{A=C}(R(A, B, O))$ b. $ir_{<AB>}(R(A, B, C))$ c. $R(A, B, C) * S(C, D, E)$ d. $R(A, B, C) \text{ US}(A, B, C)$ e. $R(A, B, C) \cap S(A, B, C)$ f. $R(A, B, C) - S(A, B, C)$ g. $R(A, B, C) \times S(D, E, F) \text{ h. } R(A, B) + S(A)$

136 Capítulo 6 A Álgebra Relacional e o Cálculo Relacional

6.31. Sugira extensões para o cálculo relacional de maneira que ele expresse os seguintes tipos de operações que foram discutidas na Seção 6.4: (a) funções agregadas e agrupamento; (b) operações de JUNÇÃO EXTERNA, e (c) consultas de clausura recursiva.

Bibliografia Seleccionada

Codd (1970) definiu a álgebra relacional básica. Date (1983a) discute junções externas. Um trabalho em operações relacionais de extensão é discutido por Carlis (1986) e Ozsoyoglu et al. (1985). Cammarata et al. (1989) estendem as restrições de integridade e junções do modelo relacional.

Codd (1971) introduziu a linguagem Alpha, que é baseada em conceitos de cálculo relacional de tupla. Alpha também inclui a noção de funções agregadas, que vai além do cálculo relacional. A definição formal original do cálculo relacional foi dada por Codd (1972), que também forneceu um algoritmo que transforma qualquer expressão de cálculo relacional de tupla em álgebra relacional. A QUEL (Stonebraker et al., 1976) é baseada no cálculo relacional de tupla, com os quantificadores existenciais implícitos, mas sem os quantificadores universais, e foi implementada no sistema Ingres como uma linguagem comercialmente disponível. Codd definiu a 'Completeza' relacional de uma linguagem de consulta que significa, no mínimo, ser tão poderosa quanto o cálculo relacional. Ullman (1988) descreve uma prova formal da equivalência da álgebra relacional com as expressões seguras de cálculo relacional de tupla e domínio. Abiteboul et al. (1995) e Atzeni e de Antonellis (1993) deram um tratamento detalhado de linguagens relacionais formais.

Embora as idéias de cálculo relacional de domínio tenham sido inicialmente propostas na linguagem QBE (Zloof, 1975), o conceito foi formalmente definido por Lacroix e Pirote (1977). A versão experimental do sistema *Query-By-Example* é descrita em Zloof (1977). A ILL (Lacroix e Pirote, 1977a) é baseada no cálculo relacional de domínio. Whang et al. (1990) estendeu a QBE com os quantificadores universais. As linguagens de consulta visuais, das quais a QBE é um exemplo, têm sido propostas como um meio de consulta a bancos de dados; conferências como o *Visual Database Systems Workshop* — por exemplo, Arisawa e Catarei (2000) ou Zhou e Pu (2002) — têm inúmeras propostas para essas linguagens.

7

Projeto de Banco de Dados Relacional pelo Mapeamento dos Modelos Entidade-Relacionamento e Entidade-Relacionamento Estendido

Vamos analisar, agora, como **projetar um esquema de um banco de dados relacional** tendo por base o esquema de um projeto conceitual. Isso corresponde ao projeto lógico do banco de dados ou ao mapeamento do modelo de dados discutido na Seção 3.1 (Figura 3.1). Apresentaremos os procedimentos para criar um esquema relacional a partir de um esquema do modelo entidade-relacionamento (ER) OU de sua extensão (EER-en/extended *entity-relationship*). Nossa discussão faz referência aos construtores dos modelos ER e EER, apresentados nos capítulos 3 e 4, e aos construtores do modelo relacional apresentados nos capítulos 5 e 6. Diversas ferramentas CASE (*computer-aided software engineering* — Engenharia de Software Apoiada por Computador) têm por base os modelos ER e EER, ou outros modelos similares, como discutimos nos capítulos 3 e 4. Essas ferramentas computadorizadas são usadas, interativamente, pelos projetistas de banco de dados, para o desenvolvimento de esquemas dos modelos ER ou EER em aplicações com banco de dados. Muitas ferramentas utilizam os diagramas ER ou EER, ou variações deles, para desenvolver os esquemas em forma de gráficos e convertê-los, automaticamente, em esquemas de um banco de dados relacional por meio da DDL de um SGBD relacional específico, empregando algoritmos similares aos apresentados neste capítulo.

Mostraremos um algoritmo com sete passos na Seção 7.1 para a conversão dos construtores básicos do modelo ER — entidades (fortes e fracas), relacionamentos binários (com diferentes restrições estruturais), relacionamentos n-ários e atributos (simples, compostos e multivalorados) — em relações. Então, na Seção 7.2, continuaremos com o algoritmo de mapeamento descrevendo como mapear os construtores do modelo EER — especialização/generalização e tipos união (categorias) — em relações.

7.1 PROJETO DE UM BANCO DE DADOS RELACIONAL. USANDO O MAPEAMENTO DO ER PARA O RELACIONAL

7.1.1 Algoritmo de Mapeamento do ER para o Relacional

Agora, vamos descrever os passos para o algoritmo de mapeamento do ER para o relacional. Vamos usar como exemplo o banco de dados EMPRESA para ilustrar os procedimentos de mapeamento. O esquema ER EMPRESA é mostrado novamente na Figura 7.1, e o esquema relacional EMPRESA correspondente, na Figura 7.2, para ilustrar os passos do mapeamento.

1 38 Capítulo 7 Projeto de Banco de Dados Relacional pelo Mapeamento dos Modelos...

supervisor

DEPENDENTE

FIGURA 7.1 Diagrama do esquema conceitual ER para o banco de dados EMPRESA.

EMPREGADO

PNOME MINICIAL UNOME SSN DATANASC ENDEREÇO SEXO SALÁRIO SUPERSN DNO

DEPARTAMENTO

DNOME DNUMERO GERSN GERDATAINICIO

S

DEPTJ,OCALIZACOES

DNUMERO DLOCALIZACAO

PNUMERO PLOCALIZACAO DNUM

TRABALHA_EM

ESSN	/ PNO	HORAS

DEPENDENTE

ESSN	NOME DEPENDENTE	SEXO	DATANASC	PARENTESCO

FIGURA 7.2 Resultado do mapeamento do esquema ER EMPRESA para o esquema de um banco de dados relacional

7.1 Projeto de um Banco de Dados Relacional Usando o Mapeamento do ER para o Relacional 1 39

Passo 1: Mapeamento dos Tipos Entidade Regulares. Para cada tipo entidade regular (forte) E de um esquema ER, criar uma relação R que inclua todos os atributos simples de E . Incluir somente os componentes simples dos atributos compostos. Escolher um dos atributos-chave de E como chave primária de R . Se a chave escolhida de E for um atributo composto, os atributos simples componentes vão, juntos, formar a chave primária de R .

Se múltiplas chaves forem identificadas para E durante o projeto conceitual, a informação que descreve os atributos que formam cada uma das chaves adicionais é apreendida pela especificação de chaves secundárias (*unique keys*) da relação R . Os conceitos sobre as chaves também são usados para os propósitos de indexação e outros tipos de análises.

Em nosso exemplo, criamos as relações EMPREGADO, DEPARTAMENTO e PROJETO na Figura 7.2 que correspondem aos tipos entidades regulares EMPREGADO, DEPARTAMENTO e PROJETO da Figura 7.1. As chaves estrangeiras e os atributos do relacionamento, se houver, não são incluídos por enquanto; eles serão contemplados nos passos subseqüentes. Isso inclui os atributos SUPERSSN e DNO de EMPREGADO, GERSSN e GERDATAINICIO de DEPARTAMENTO, e DNUM de PROJETO. Em nosso exemplo, escolhemos SSN, DNUMERO e PNUMERO como chaves primárias para as relações EMPREGADO, DEPARTAMENTO e PROJETO, respectivamente. O conhecimento de que DNOME de DEPARTAMENTO e de que PJNOME de PROJETO são chaves secundárias é guardado para garantir possíveis usos futuros no projeto.

As relações que são criadas pelo mapeamento dos tipos entidade são, algumas vezes, chamadas relações entidades, pois cada tupla (linha) representa uma instância de uma entidade.

Passo 2: Mapeamento dos Tipos Entidade Fracas. Para cada tipo entidade fraca W de um esquema ER, que contém sua respectiva entidade forte E , criar uma relação R e nela incluir todos os atributos simples (ou os componentes simples dos atributos compostos) de W como atributos de R . Além disso, inserir como chave estrangeira de R os atributos que são chaves primárias da(s) relação(ões) que corresponde(m) ao mapeamento do tipo entidade(s) forte(s) correspondente(s); isso identifica o(s) tipo(s) relacionamento(s) de W . A chave primária de R é a combinação da(s) chave(s) primária(s) da(s) forte(s) e da chave parcial do tipo entidade fraca W , se houver.

Se existir um tipo entidade fraca E_2 cuja entidade forte é também um tipo entidade fraca E_1 , então E_1 deveria ser mapeada antes de E_2 , de modo a determinar, primeiro, sua chave primária.

Em nosso exemplo, criamos a relação DEPENDENTE nesse passo para o mapeamento do tipo entidade fraca DEPENDENTE. Incluímos a chave primária SSN da relação EMPREGADO — que corresponde ao tipo entidade forte — como chave estrangeira de DEPENDENTE, e renomeamos esse atributo como ESSN, embora isso não seja obrigatório. A chave primária da relação DEPENDENTE é a combinação {ESSN, NOME_DEPENDENTE}, porque NOME_DEPENDENTE é chave parcial de DEPENDENTE.

É comum escolher a opção de propagação (CASCADE) para as ações de gatilhos (*triggers*) definidas (Seção 8.2) sobre as chaves estrangeiras da relação correspondente ao tipo entidade fraca, uma vez que a existência da entidade fraca depende de sua entidade forte. Essa opção pode ser usada em ambas as funções ON UPDATE (atualização) e ON DELETE (exclusão).

Passo 3: Mapeamento dos Tipos Relacionamento Binários 1:1. Para cada relacionamento R binário 1:1 em um esquema ER, identificar as relações S e T que correspondem aos tipos entidades participantes de R . Há três escolhas possíveis: (1) a da chave estrangeira; (2) a do relacionamento incorporado; e (3) a da referência cruzada ou da relação de relacionamento. A primeira opção é a mais usada e deve ser seguida, exceto se houver condições especiais, que serão discutidas a seguir.

1. *Escolha da chave estrangeira*: escolher uma das relações — S , digamos — e nela inserir, como chave estrangeira, a chave primária de T . É melhor escolher, entre os tipos entidade, aquele com a *participação total* em R para exercer o papel de S . Incluir todos os atributos simples (ou os componentes simples dos atributos compostos) do tipo relacionamento 1:1 R como atributos de S . Em nosso exemplo, mapeamos o tipo relacionamento 1:1 GERENCIA da Figura 7.1, escolhendo o tipo entidade participante DEPARTAMENTO para exercer o papel de S , pois sua participação no tipo relacionamento GERENCIA é total (todo departamento possui um gerente). Inserimos a chave primária da relação EMPREGADO como chave estrangeira na relação DEPARTAMENTO e a denominamos GERSSN. Também inserimos o atributo simples DATAINICIO, pertencente ao tipo relacionamento GERENCIA, na relação DEPARTAMENTO e o denominamos GERDATAINICIO.

Observe que é possível, como alternativa, inserir a chave primária de S como chave estrangeira de T . Em nosso exemplo, seria o equivalente a ter um atributo como chave estrangeira, digamos DEPARTAMENTO0J3ERENCIAD0, na relação EMPREGADO, mas esse atributo assumiria valor *null* (ausência de valor) nas tuplas dos empregados que não fossem gerentes de departamento. Se apenas 10% dos empregados gerenciassem departamentos, então 90% dos valores dessa chave estrangeira assumiriam valores *null*. Outra possibilidade seria ter chaves estrangeiras em ambas as relações S e T de modo redundante, mas isso incorreria possíveis prejuízos de consistência na manutenção.

140 Capítulo 7 Projeto de Banco de Dados Relacional pelo Mapeamento dos Modelos...

2. *Opção da relação unificada*: uma possibilidade para o mapeamento de um tipo relacionamento 1:1 é incorporar o tipo relacionamento e os dois tipos entidade envolvidos a uma única relação. Essa opção pode ser apropriada quando *ambas as participações são totais*.

3. *Opção referência cruzada ou relação de relacionamento*: a terceira opção é definir uma terceira relação *R* com a finalidade de servir de referência cruzada às chaves primárias das duas relações *S* e *T* que mapeiam os tipos entidade. Como veremos adiante, essa alternativa é necessária para o mapeamento de relacionamentos binários *N:M*. A relação *R* é chamada relação de relacionamento (ou, algumas vezes, tabela de busca), porque cada tupla em *R* representa uma instância do relacionamento, que relaciona uma tupla de *S* a uma tupla de *T*.

Passo 4: Mapeamento dos Tipos Relacionamento Binário 1:N. Para cada tipo relacionamento *R* binário 1:N regular, identificar a relação *S* que representa o tipo entidade participante do lado *N* do tipo relacionamento. Inserir em *S*, como chave estrangeira, a chave primária da relação *T* que representa o outro tipo entidade participante em *R*; isso é feito porque cada instância de entidade do lado *N* do tipo relacionamento está relacionada a, no máximo, uma instância do lado 1. Incluir qualquer atributo simples (ou componentes simples de atributos compostos) do tipo relacionamento 1:N como atributo de *S*.

Em nosso exemplo, mapearemos agora os tipos relacionamento 1:N TRABALHA_PARA, CONTROLE e SUPERVISÃO da Figura 7.1. Para TRABALHA_PARA, vamos inserir a chave primária DNUMERO da relação DEPARTAMENTO como chave estrangeira da relação EMPREGADO e a chamaremos DN0. Para SUPERVISÃO incluiremos a chave primária da relação EMPREGADO como chave estrangeira na própria relação EMPREGADO — isso porque o relacionamento é recursivo — e a denominaremos SUPERSSN. O relacionamento CONTROLE é mapeado pela chave estrangeira DNUM em PROJETO, a qual faz referência à chave primária DNUMERO da relação DEPARTAMENTO.

Uma alternativa que poderíamos usar aqui seria novamente criar uma relação relacionamento (referência cruzada), como vimos no caso dos relacionamentos binários 1:1. Criaríamos uma relação *R*, separada, cujos atributos seriam as chaves de *S* e *T*, e cuja chave primária seria a mesma chave de *S*. Essa escolha pode ser usada se poucas tuplas de *S* participarem do relacionamento de forma a evitar o excesso de valores *null* na chave estrangeira.

Passo 5: Mapeamento dos Tipos Relacionamento Binário N:M. Para cada tipo relacionamento *R* binário *N:M*, criar uma nova relação *S* para representar *R*. Inserir, como chave estrangeira em *S*, as chaves primárias das relações que representam os tipos entidade participantes do relacionamento; a combinação (concatenação) delas formará a chave primária de *S*. Também devem ser incluídos quaisquer atributos simples do tipo relacionamento *N:M* (ou os componentes simples dos atributos compostos) como atributos de *S*. Observe que não podemos representar um tipo relacionamento *N:M* por meio de uma simples chave estrangeira em uma das relações participantes (como fizemos para os tipos relacionamentos 1:1 e 1:N) por causa da razão de cardinalidade *N:M*; precisamos criar uma *relação relacionamento S* separada.

Em nosso exemplo, mapeamos o tipo relacionamento *N:M* TRABALHA_PARA, da Figura 7-1, criando a relação TRABALHA_PARA da Figura 7.2. Inserimos as chaves primárias das relações PROJETO e EMPREGADO como chaves estrangeiras em TRABALHA_PARA, denominadas PNO e ESSN, respectivamente. Também incluímos o atributo HORAS em TRABALHA_PARA para representar o atributo HORAS do tipo relacionamento. A chave primária da relação TRABALHA_PARA é a combinação das chaves estrangeiras {ESSN, PNO}.

A opção de propagação (CASCADE) para as ações de gatilhos (*trigger*) associadas (Seção 8.2) poderia ser especificada para as chaves estrangeiras da relação correspondente ao relacionamento *R*, uma vez que cada instância de relacionamento depende, para existir, de cada entidade a ela relacionada. Isso pode ser usado para ambas as funções ON UPDATE (atualização) e ON DELETE (exclusão).

Observe que poderemos sempre mapear os relacionamentos 1:1 e 1:N de modo similar ao relacionamento *N:M* usando a referência cruzada (relação relacionamento), conforme discutimos anteriormente. Essa alternativa é particularmente interessante quando existem poucas instâncias do relacionamento, evitando os valores *null* nas chaves estrangeiras. Nesse caso, a chave primária da relação relacionamento será *apenas uma* das chaves estrangeiras que fazem referência às relações das entidades participantes. Para um relacionamento 1:N, a chave primária da relação relacionamento será a chave estrangeira que faz referência à relação da entidade do lado *N*. Para um relacionamento 1:1, cada uma das chaves estrangeiras pode ser usada como chave primária da relação relacionamento à medida que nenhuma entrada *null* será possível nessa relação.

Passo 6: Mapeamento de Atributos Multivalorados. Para cada atributo multivalorado *A*, criar uma nova relação *R*. Essa relação deverá conter um atributo correspondente a *A*, mais a chave primária *K* — como chave estrangeira em *R* — da relação que representa o tipo entidade ou o tipo relacionamento que tem *A* como atributo. A chave primária de *R* é a combinação de *A* e *K*. Se o atributo multivalorado for um atributo composto, incluiremos seus componentes simples.

Em nosso exemplo, criamos uma relação DEPT_LOCALIZACOES. O atributo DLOCALIZACAO representa o atributo multivalorado LOCALIZAÇÕES de DEPARTAMENTO, enquanto DNUMERO — como chave estrangeira — representa a chave primária da relação DEPARTAMENTO.

7.1 Projeto de um Banco de Dados Relacional Usando o Mapeamento do ER para o Relacional 141

A chave primária de DEPT_LOCAUZACOES é a combinação de {DNUMERO, DLOCALIZACAO}. Existirá uma tupla separada em DEPT_LOCALIZACOES para cada localização que um departamento tiver.

As opções de propagação (CASCADE) para as ações de gatilhos (trigger) associadas (Seção 8.2) poderiam ser especificadas para a chave estrangeira da relação R correspondente ao atributo multivalorado tanto para ONJPPDATE (atualização) quanto para ON_DELETE (exclusão). Devemos notar que a chave de R, quando no mapeamento de atributos compostos e mul-tivalorados, exige alguma análise do significado dos atributos componentes. Em alguns casos, quando um atributo multivalorado é composto, somente alguns desses atributos componentes são necessários para compor a chave de R; esses atributos são similares a uma chave parcial de um tipo entidade fraca que corresponde ao atributo multivalorado (Seção 3.5).

A Figura 7.2 apresenta o esquema do banco de dados relacional EMPRESA, obtido pela seqüência dos passos 1 a 6, e a Figura 5.6 mostra um exemplo de banco de dados instanciado. Observe que ainda não discutimos o mapeamento dos tipos relacionamentos n-ários (n>2) porque não existe nenhum na Figura 7.1; eles são mapeados de modo similar aos tipos relacionamentos N:M, por meio da inclusão do seguinte passo adicional no algoritmo de mapeamento.

Passo 7: Mapeamento dos Tipos Relacionamento N-ário. Para cada tipo relacionamento R n-ário, em que n > 2, criar uma nova relação S para representar R. Incluir, como chave estrangeira em S, as chaves primárias das relações que representam os tipos entidade participantes. Incluir, também, qualquer atributo simples do tipo relacionamento n-ário (ou os componentes simples dos atributos compostos) como atributo de S. A chave primária de S é, normalmente, a combinação de todas as chaves estrangeiras que fazem referência às relações representantes dos tipos entidade participantes. Entretanto, se as restrições de cardinalidade em qualquer um dos tipos entidade E participantes em R forem 1, então a chave primária de S não deve incluir a chave estrangeira que faz referência à relação E' correspondente a E (Seção 4.7).

Por exemplo, considere o tipo relacionamento FORNECE da Figura 4-11 a. Ele pode ser mapeado pela relação FORNECE, mostrada na Figura 7.3, cuja chave primária é a combinação das três chaves estrangeiras (FNOME, NUMLOTE, NOMEPROJ).

7.1.2 Discussão e Resumo do Mapeamento dos Construtores dos Modelos

A Tabela 7.1 resume a correspondência entre os construtores e as restrições dos modelos ER e relacional.

Um dos principais pontos que deve ser notado em um esquema relacional, em contraste com um esquema ER, é que os tipos relacionamento não são representados explicitamente; pelo contrário, são representados por dois atributos A e B, um a chave primária, e outro, a chave estrangeira (sobre o mesmo domínio), inseridos em duas relações S e T. Duas tuplas em S e T são relacionadas quando têm o mesmo valor em A e B. Pelo uso da operação EQUIJOIN (ou NATURAL JOIN — junção natural —, se os dois atributos de junção possuírem o mesmo nome) em S.A e T.B, podemos combinar todos os pares de tuplas relacionados entre S e T e, assim, efetivar o relacionamento. Quando um tipo relacionamento 1:1 ou 1:N está envolvido, normalmente uma única operação de junção é suficiente. Para um tipo relacionamento binário N:M, duas operações de junção são necessárias, ao passo que, nos tipos relacionamentos n-ários, n junções são essenciais para materializar completamente as instâncias do relacionamento.

FORNECEDOR

FNOME	
-------	--

PROJETO

NOMEPROJ	• • •
----------	-------

LOTE

NUMLOTE	
---------	--

FORNECE

FNOME	NOMEPROJ	NUMLOTE	QUANTIDADE

FIGURA 7.3 Mapeamento do tipo relacionamento n-ário FORNECE, da Figura 4.11 a.

Modelo ER

Modelo Relacional

Tipo entidade

Tipo relacionamento 1:1 ou 1:N Tipo relacionamento N:M Tipo relacionamento n-ário Atributo simples Atributo composto Atributo multivalorado Conjunto de valores Atributo-chave

Relação 'entidade'

Chave estrangeira (ou relação 'relacionamento')

Relação 'relacionamento' e duas chaves estrangeiras

Relação 'relacionamento' e n chaves estrangeiras

Atributo

Conjunto de atributos simples componentes

Relação e chave estrangeira

Domínio

Chave primária (ou secundária)

Por exemplo, para formar uma relação com o nome do empregado, o nome do projeto e as horas que o empregado trabalhou em cada projeto, precisamos conectar cada tupla de EMPREGADO às tuplas de PROJETO a ela relacionadas, por intermédio da relação TRABALHA_EM da Figura 7.2. Então precisamos aplicar a operação EQUIJOIN nas relações EMPREGADO e TRABALHA_EM usando como condição de junção $SSN = ESSN$, e então aplicar outra operação EQUIJOIN entre a relação resultante e a relação PROJETO utilizando como condição de junção $PNO = PNUMERO$. Geralmente, quando muitos relacionamentos são utilizados, é necessário especificar as diversas operações de junção. O usuário de um banco de dados relacional precisa sempre tomar cuidado com os atributos empregados como chave estrangeira, de forma a combiná-los corretamente em tuplas relacionadas a duas ou mais relações. Muitas vezes, esse problema é considerado um inconveniente do modelo relacional, uma vez que a correspondência entre as chaves primárias/estrangeiras nem sempre é evidente quando se examina um esquema relacional. Se um *equijoin* é realizado sobre os atributos de duas relações, e se esses atributos não correspondem à chave primária/chave estrangeira de um relacionamento, o resultado frequentemente não tem sentido e pode apresentar dados falsos (inválidos). Por exemplo, o leitor pode desejar juntar as relações PROJETO e DEPT_L0CAUZAC0ES por meio da condição $DLOCAUZACAO = PLOCALIZACAO$ e examinar o resultado (Capítulo 10).

Outro ponto que merece atenção nos esquemas relacionais é que criamos uma relação separada para *cada* atributo multivalorado. Para uma entidade em particular, com um conjunto de valores como atributos multivalorados, o valor do atributo-chave da entidade é repetido uma vez para cada valor do atributo multivalorado da tupla da relação separada. Isso porque o modelo relacional básico *não* permite os valores múltiplos (uma lista ou um conjunto de valores) para um atributo em uma única tupla. Por exemplo, como o departamento 5 tem três localizações, existem três tuplas na relação DEPT_LOCAUZACOES da Figura 5.6; cada tupla especifica uma de suas localizações. Em nosso exemplo, aplicamos EQUIJOIN para DEPT_L0CALIZAC0ES e DEPARTAMENTO sobre o atributo DNUMERO para obter os valores de todas as localizações com todos os demais atributos de DEPARTAMENTO. Na relação resultante, os valores dos demais atributos do departamento são repetidos em uma tupla separada para cada uma das localizações que um departamento tiver.

A álgebra relacional básica não contém as operações NEST e COMPRESS, que poderiam produzir, a partir da relação DEPTJ.0CAUZAC0ES da Figura 5.6, um conjunto de tuplas da forma $\{<1, Houston>, <4, Stafford>, <5, \{Bellaire, Sugarland, Hous-ton\}>\}$. Essa é uma desvantagem séria da normalização básica ou versão '*flat*' do modelo relacional. Nesse sentido, o modelo orientado a objeto e os modelos legados hierárquico e de rede oferecem mais facilidades que o modelo relacional. O modelo relacional aninhado (nested) e os sistemas objeto-relacional (Capítulo 22) tentam minimizar esse problema.

7.2 MAPEAMENTO DOS CONSTRUTORES DO MODELO EER PARA AS RELAÇÕES

Vamos discutir agora o mapeamento dos construtores do modelo EER para as relações por meio da extensão do algoritmo para o mapeamento do modelo ER para o relacional, apresentado na Seção 7.1.1.

7.2.1 Mapeamento da Especialização ou Generalização

Há diversas opções para o mapeamento de um grupo de subclasses que, juntas, formam uma especialização (ou, alternativamente, são generalizadas em uma superclasse), como {SECRETARIA, TÉCNICO, ENGENHEIRO}, que são as subclasses de EMPREGADO na Figura 4.4. Podemos adicionar um passo ao algoritmo de mapeamento do ER para o relacional da Seção 7.1.1, o qual possui sete pas-

7.2 Mapeamento dos Construtores do Modelo EER para as Relações 143

so, de modo a contemplar o mapeamento da generalização. O Passo 8, apresentado a seguir, fornece as opções mais comuns; outros mapeamentos são também possíveis. Vamos discutir, então, sob quais condições cada uma dessas opções deveria ser preferida. Usamos $Atr(R)$ para designar os *atributos da relação* R e $PK(R)$ para designar a *chave primária de* R .

Passo 8: Opções para o Mapeamento da Especialização ou Generalização. Converter cada especialização com m subclasses $\{S_1, S_2, \dots, S_m\}$ e a superclasse C (generalizada), em que os atributos de C são $\{k, a_1, a_2, \dots, a_n\}$ e k é chave (primária) em esquemas de relações usando uma das quatro opções seguintes:

- Opção 8A: *relações múltiplas — superclasse e subclasse*. Criar uma relação L para C com os atributos $Atr(L) = \{k, a_1, a_2, \dots, a_n\}$ e $PK(L) = k$. Criar uma relação L_i para cada subclasse S_i , $1 < i < m$, com os atributos $Atr(L_i) = \{k\} \cup \{\text{atributos de } S_i \text{ e } PK(L_i) = k\}$. Essa opção funciona para qualquer especialização (total ou parcial, disjuntas ou sobrepostas).
- Opção 8B: *relações múltiplas — somente relações de subclasses*. Criar uma relação L_i para cada subclasse S_i , $1 < i < m$, com os atributos $Atr(L_i) = \{\text{atributos de } S_i \cup \{k, a_1, a_2, \dots, a_n\}\}$ e $PK(L_i) = k$. Essa opção funciona somente para as especializações cujas subclasses são *totais* (toda entidade em uma superclasse deve pertencer a [pelo menos] uma subclasse).
- Opção 8C: *relação única com um atributo tipo*. Criar uma única relação L com os atributos $Atr(L) = \{k, a_1, a_2, \dots, a_n\} \cup \{t\}$ e $PK(L) = k$. O atributo t é chamado atributo tipo (ou discrimi-nativo), que indica a subclasse à qual cada tupla pertence, se pertencer a alguma. Essa opção funciona para as especializações cujas subclasses sejam *disjuntas* e tenham potencial para gerar muitos valores *null* se houver diversos atributos específicos nas subclasses.
- Opção 8D: *relação única com o tipo atributos múltiplo*. Criar uma única relação esquema L com os atributos $Atr(L) = \{k, a_1, a_2, \dots, a_n\} \cup \{\text{atributos de } S_1 \cup \dots \cup \{\text{atributos de } S_m\}\}$ e $PK(L) = k$. Cada t_i , $1 < i < m$, é um atributo do tipo booleano indicando se a tupla pertence ou não àquela subclasse S_i . Essa opção funciona para as especializações cujas subclasses sejam *sobrepostas* (embora também funcione para as especializações disjuntas).

As opções 8A e 8B podem ser chamadas opções de relações-múltiplas, enquanto as opções 8C e 8D podem ser chamadas opções de relações-simples. A Opção 8A cria uma relação L para a superclasse C e seus atributos, mais uma relação L_i para cada subclasse S_i ; cada L_i engloba os atributos específicos (ou locais) de S_i mais a chave primária da superclasse C , que é propagada para L_i e torna-se sua chave primária. Uma operação EQUIJOIN, igualando a chave primária entre qualquer L_i e L , produz todos os atributos específicos e inerentes das entidades em S_i . Essa opção é ilustrada na Figura 7.4a para o esquema EER da Figura 4.4. A Opção 8A funciona para qualquer condicionante da especialização: disjunta ou sobreposta, total ou parcial. Observe que a condição

$ir_{C \rightarrow L} \cap \bigcap_{i=1}^m ir_{S_i \rightarrow L}$

deve ser assegurada para cada L_i . Isso especifica uma chave estrangeira de L para cada L_i , bem como a *dependência de inclusão* $L_i, k < L, k$ (Seção 11.5).

Na Opção 8B, a operação de EQUIJOIN é *construída dentro* do esquema e a relação L é eliminada, como ilustrado na Figura 7.4b para a especialização EER da Figura 4.3b. Essa opção funciona bem somente quando há *ambas* as especializações, disjuntas e totais. Se a especialização não for total, uma entidade que não pertença a nenhuma subclasse S_i estará perdida. Se as especializações não forem disjuntas, uma entidade que pertencer a mais de um tipo de subclasse terá seus atributos inerentes provenientes da superclasse C armazenados de modo redundante em mais de uma L_i . Na Opção 8B, nenhuma relação mantém todas as entidades da superclasse C ; conseqüentemente, precisamos aplicar uma operação OUTER UNION (ou FULL OUTER JOIN) nas relações L_i para recuperar todas as entidades de C . O resultado da junção externa (*outerjoin*) será similar às relações das opções 8C e 8D, com exceção de que os campos tipo serão omitidos. Toda vez que pesquisarmos uma entidade qualquer de C precisaremos consultar todas as m relações L_i .

As opções 8C e 8D criam uma relação única para representar a superclasse C e todas as suas subclasses. Uma entidade que não pertença a nenhuma subclasse terá valores *null* para os atributos específicos dessa subclasse. Portanto, essas opções não são recomendadas para o mapeamento quando houver muitos atributos específicos das subclasses. Entretanto, se existirem poucos atributos específicos nas subclasses, esses mapeamentos são preferíveis às opções 8A e 8B, porque eliminam a necessidade de especificar operações de EQUIJOIN e OUTERJOIN produzindo as implementações mais eficientes.

A Opção 8C é usada para o tratamento das subclasses disjuntas pela inclusão de um único atributo tipo (ou imagem ou descritivo) t para indicar a subclasse à qual cada tupla pertence; logo, seu domínio poderia ser $\{1, 2, \dots, m\}$. Se a especialização for parcial, t poderá ter valores *null* nas tuplas que não pertençam a nenhuma subclasse. Se a especialização for atributo-definida, o atributo definirá o propósito de t , e t não será necessário; a Figura 7.4c ilustra essa opção para a especialização EER da Figura 4.4.

144 Capítulo 7 Projeto de Banco de Dados Relacional pelo Mapeamento dos Modelos.

(a) EMPREGADO

SSN	PNome	MInicial	UNome	DataNasc	Endereço	TipoTrabalho
-----	-------	----------	-------	----------	----------	--------------

SECRETARIA

TÉCNICO

ENGENHEIRO

SSN VelocidadeDigitacao

SSN	TGrau
-----	-------

SSN	TipoEng
-----	---------

(b) CARRO

IdVeiculo	NrLicencaPlaca	Preço	VelocidadeMax	NrDePassageiros
-----------	----------------	-------	---------------	-----------------

CAMINHÃO

IdVeiculo	NrLicencaPlaca	Preço	NrDeEixos	Capacidade
-----------	----------------	-------	-----------	------------

(c) EMPREGADO

SSN	PNome	MInicial	UNome	DataNasc	Endereço	TipoTrabalho	VelocidadeDigitacao	TGrau	TipoEng
-----	-------	----------	-------	----------	----------	--------------	---------------------	-------	---------

(d) PEÇA

NumPeca	Descrição	MFlag	NumDesenho	DataFabricacao	NumLote	PFlag	NomeFornecedor	ListaPreco
---------	-----------	-------	------------	----------------	---------	-------	----------------	------------

FIGURA 7.4 Opções para o mapeamento da especialização ou generalização, (a) Mapeamento do esquema EER da Figura 4.4 usando-se a Opção 8A. (b) Mapeamento do esquema EER da Figura 4.3 utilizando-se a Opção 8B. (c) Mapeamento do esquema EER da Figura 4.4 empregando-se a Opção 8C. (d) Mapeamento da Figura 4.5 aplicando-se a Opção 8D com os campos do tipo booleano MFlag e PFlag.

A Opção 8D é indicada para o tratamento das subclasses sobrepostas que incluem um campo tipo *booleano m* para cada subclasse. Também pode ser usada para as subclasses disjuntas. Cada campo tipo t_i pode ter domínio {sim, não}, no qual o valor sim indica que a tupla é membro da subclasse S_i . Se usarmos essa opção para a especialização EER da Figura 4.4, poderemos inserir três tipos de atributos — EUmaSecretaria, EUmEngenheiro e EUmTécnico — em vez do atributo TipoTrabalho da Figura 7.4c. Observe que também é possível criar um atributo tipo único de m bits, em vez de m campos tipo.

Quando temos uma especialização (ou generalização) em vários níveis hierárquicos ou de reticulado (*lattice*), não temos de seguir a mesma opção de mapeamento para todas as especializações. Podemos usar uma opção para o mapeamento de uma parte da hierarquia ou do reticulado e a outra opção para a outra parte. A Figura 7.5 mostra uma possibilidade de mapeamento, em relações, do reticulado EER da Figura 4-6. Usamos aqui a Opção 8A para PESSOA/{EMPREGADO, EXALUNO, ALUNO}, a Opção 8C para EMPREGADO/{FUNCIONARIO, DOCENTE, ASSISTENTE_ALUNO} e a Opção 8D para ASSISTENTE_ALUNO/{ASSISTENTE_PESQUISA, ASSISTENTE_ENSINO} ALUNO/ASSISTENTE_ALUNO (em ALUNO), e ALUNO/{ALUNO_GRADUADO, ALUNO_NAO_GRADUADO}. Na Figura 7.5, todos os atributos cujos nomes começam com 'Tipo' ou terminam com 'Flag' são campos tipo.

PESSOA

SSN	Nome	DataNasc	Sexo	Endereço
-----	------	----------	------	----------

EMPREGADO

SSN	Salário	TipoEmpregado	Funcao	Categoria	PorcentagemTempo	RAFlag	TAFlag	Projeto	Curso
-----	---------	---------------	--------	-----------	------------------	--------	--------	---------	-------

EXALUNO FORMACAO_EXALUNO

SSN

SSN	Ano	Formação	Habilitação
-----	-----	----------	-------------

ALUNO

SSN	DeptHab	FormFlag	NFormFlag	ProgramaTitulacao	Classe	AlunoAssisFlag
-----	---------	----------	-----------	-------------------	--------	----------------

FIGURA 7.5 Mapeamento do reticulado de especialização EER da Figura 4.6 usando-se diversas opções.

7.2 Mapeamento dos Construtores do Modelo EER para as Relações

145

7.2.2 Mapeamento de Subclasses Compartilhadas (Herança Múltipla)

Uma subclasse compartilhada, como GERENTE_ENGENHARIA da Figura 4.6, é a subclasse de diversas superclasses, indicando uma herança múltipla. Essas classes devem ter o mesmo atributo-chave, porém, as subclasses compartilhadas poderiam ser modeladas como categorias. Podemos aplicar quaisquer opções apresentadas no Passo 8 para as subclasses compartilhadas, guardadas as condicionantes discutidas no Passo 8 para o algoritmo de mapeamento. Na Figura 7.5, ambas as opções 8C e 8D são usadas para as subclasses compartilhadas ASSISTENTE_ALUNO. A Opção 8C é utilizada na relação EMPREGADO (atributo TipoEmpregado) e a Opção 8D é empregada na relação ALUNO (atributo AlunoAssisFlag).

7.2.3 Mapeamento de Categorias (Tipos União)

Vamos, desta vez, adicionar outro passo — o Passo 9 — ao processo de mapeamento para o tratamento de categorias (agregação). Uma categoria (ou tipo união) é uma subclasse da *união* de duas ou mais superclasses que podem ter chaves diferentes, uma vez que pode haver diferentes tipos entidades. Um exemplo é a categoria PROPRIETÁRIO, mostrada na Figura 4-7, que é um subconjunto da união de três tipos entidade: PESSOA, RANÇO e EMPRESA. A outra categoria dessa figura, VEICULO_REGISTRADO, tem duas superclasses que possuem o mesmo atributo-chave.

Passo 9: Mapeamento dos Tipos União (Categorias). Para o mapeamento de uma categoria que define as superclasses que têm chaves diferentes, é praxe especificar um novo atributo-chave, chamado **chave substituta**, ao se criar a relação correspondente à categoria. Isso porque as chaves das classes estabelecidas são diferentes, de modo que não podemos usar nenhuma delas em particular para a identificação de todas as entidades da categoria. Em nosso exemplo da Figura 4.7, podemos criar uma relação PROPRIETÁRIO que corresponde à categoria PROPRIETÁRIO, como ilustrado na Figura 7.6, e incluir qualquer um dos atributos da categoria nessa relação. A chave primária da relação PROPRIETÁRIO é a chave substituta, que é chamada IdProprietario. Também incluiremos a chave substituta IdProprietario como chave estrangeira em cada relação correspondente à superclasse da categoria, de modo a especificar a correspondência em valores entre a chave substituta e a chave de cada superclasse. Observe que, se uma entidade em particular PESSOA (ou BANCO ou EMPRESA) não for um membro de PROPRIETÁRIO, poderia haver valor *null* para o atributo IdProprietario em suas tuplas correspondentes nas relações PESSOA (ou BANCO ou EMPRESA), e poderia não haver uma tupla na relação PROPRIETÁRIO.

PESSOA

SSN	NrLicencaMotorista	Nome	Endereço	IdProprietario
-----	--------------------	------	----------	----------------

BANCO

NomeB EnderecoB IdProprietario

EMPRESA

NomeE	EnderecoE	IdProprietario
-------	-----------	----------------

PROPRIETÁRIO

IdProprietario

VEICULO_REGISTRADO

IdVeiculo	NrLicencaPlaca
-----------	----------------

CARRO

IdCarro	EstiloC	MarcaC	ModeloC	AnoC
---------	---------	--------	---------	------

CAMINHÃO

IdCam	MarcaCam	ModeloCam	Capacidade	AnoCam
-------	----------	-----------	------------	--------

POSSUI

IdProprietario	IdVeiculo	DataAquisicao	AlienadoOuRegular
----------------	-----------	---------------	-------------------

FIGURA 7.6 Mapeamento, em relações, das categorias (tipos união) EER da Figura 4.7.

146 Capítulo 7 Projeto de Banco de Dados Relacional pelo Mapeamento dos Modelos...

Para uma categoria cujas superclasses têm a mesma chave, como VEICULO da Figura 4.7, não existe nenhuma necessidade da chave substituta. O mapeamento da categoria VEICULO_REGISTRADO, que ilustra esse caso, é também mostrado na Figura 7.6.

7.3 RESUMO

Na Seção 7.1, mostramos como um projeto de um esquema conceitual do modelo ER pode ser mapeado para um esquema de um banco de dados relacional. Foi fornecido e ilustrado, por meio dos exemplos do banco de dados EMPRESA, um algoritmo de mapeamento do ER para o relacional. A Tabela 7.1 resume as correspondências entre os construtores e as restrições do modelo ER e o relacional. Adicionamos, então, a esse algoritmo da Seção 7.2, os passos necessários para o mapeamento dos construtores do modelo EER para o modelo relacional. Os algoritmos similares são incorporados a ferramentas de projetos gráficos de um banco de dados para automaticamente criar um esquema relacional a partir de um projeto de um esquema conceitual.

Questões para Revisão

7.1. Discuta a correspondência entre os construtores do modelo ER e os do modelo relacional. Mostre como cada construtor do modelo ER pode ser mapeado para o modelo relacional e discuta as alternativas de mapeamento.

7.2. Discuta as opções para o mapeamento dos construtores EER para as relações.

Exercícios

7.3. Tente mapear o esquema relacional da Figura 6.12 para um esquema ER. Essa é uma parte do processo conhecida como *engenharia reversa*, na qual um esquema conceitual é criado a partir de um banco de dados implementado. Fundamente qualquer pressuposto que você faça.

7.4. A Figura 7.7 revela um esquema ER para um banco de dados que pode ser usado para o monitoramento de navios de carga, com suas localizações para as autoridades marítimas. Mapeie esse esquema para um esquema relacional e especifique todas as chaves primárias e estrangeiras.

FIGURA 7.7 Um esquema ER para o banco de dados MONITORAMENTO_NAVIO.

7.3 Resumo | 147

7.5. Mapeie o esquema ER BANCO do Exercício 3.23 (mostrado na Figura 3.17) em um esquema relacional. Especifique todas as chaves primárias e estrangeiras. Repita o procedimento para o esquema COMPANHIA AÉREA (Figura 3.16) do Exercício 3.19 e para os demais esquemas dos exercícios 3.16 ao 3.24.

7.6 Mapeie os diagramas EER das figuras 4.10 e 4-17 em esquemas relacionais. Justifique suas escolhas dentre as opções de mapeamento.

Bibliografia Seleccionada

O algoritmo de mapeamento ER para o relacional original foi descrito por Chen em seu artigo clássico (Chen, 1976), que apresentou o modelo ER original.

8

SQL-99: Definição de Esquema, Restrições Básicas e Consultas *{Queries}*

A linguagem SQL pode ser considerada uma das maiores razões para o sucesso dos bancos de dados relacionais no mundo comercial. Como se tornou padrão para os bancos relacionais, os usuários têm pouca preocupação ao migrar suas aplicações de banco de dados, originadas por outros tipos de sistemas de banco de dados — por exemplo, em rede e hierárquico —, para o sistema relacional. A razão é que, mesmo que se sintam insatisfeitos com o SGBD relacional que decidiram usar, qualquer que seja, não se espera que a conversão para outro SGBD relacional gere custo e tempo excessivos, uma vez que ambos os sistemas devem seguir os mesmos padrões de linguagem. Na prática, naturalmente, existem muitas diferenças entre os SGBDs relacionais comerciais. Entretanto, se o usuário for diligente e usar somente aquelas funcionalidades que fazem parte do padrão, e se ambos os sistemas suportarem plenamente esse padrão, então a conversão entre os dois sistemas será muito fácil. Outra vantagem de possuir esse padrão é que os usuários podem escrever declarações em um programa aplicativo e podem acessar dados armazenados em dois ou mais SGBDs relacionais, sem ter de alterar a sublinguagem de banco de dados (SQL) em ambos SGBDs.

Este capítulo apresenta as principais funcionalidades do padrão SQL para os SGBDs relacionais *comerciais*, enquanto o Capítulo 5 apresenta os conceitos mais importantes que fundamentam o modelo *formal* de um banco de dados relacional. No Capítulo 6 (seções 6.1 a 6.5), discutimos as operações da *álgebra relacional*, que é muito importante para o entendimento dos tipos de requisitos que podem ser especificados em um banco de dados relacional. Essas operações também são importantes para o processamento e para a otimização de consultas em um SGBD relacional, como será visto nos capítulos 15 e 16. Entretanto, as operações da álgebra relacional são consideradas muito técnicas para a maioria dos usuários dos SGBDs relacionais comerciais, porque uma consulta em álgebra relacional é escrita como uma seqüência de operações que, quando executadas, produzem o resultado desejado. Assim, o usuário precisa especificar como — ou melhor, em *qual ordem* — as operações de uma consulta devem ser executadas. No entanto, a linguagem SQL proporciona uma interface *declarativa* de alto nível, então, o usuário somente especifica *qual* será o resultado, deixando com o SGBD a otimização e a decisão de como, de fato, executar a consulta. Embora a SQL inclua apenas as funcionalidades da álgebra relacional, em grande parte ela tem base no *cálculo relacional de tuplas*, descrito na Seção 6.6. Entretanto, a sintaxe da SQL é mais amigável que qualquer uma das duas linguagens formais.

O nome da SQL é derivado de *Structure Query Language* (Linguagem Estruturada de Consulta), e foi chamada inicialmente SEQUEL (*Structured English QUERY Language* — Linguagem de Pesquisa em Inglês Estruturado), sendo projetada e implementada na IBM Research como uma interface para um sistema experimental de um banco de dados relacional chamado SISTEMA R. A SQL é agora a linguagem-padrão para os SGBDs relacionais comerciais. Um esforço conjunto da ANSI (*American National Standards Institute* — Instituto Nacional Americano de Padrões) e da ISO (*International Standards Organization* — Organização Internacional de Padrões) chegou à versão-padrão da SQL (ANSI, 1986), chamada SQL-86 ou SQL1. Uma versão revisada e expandida chamada SQL2 (também conhecida como SQL-92) foi desenvolvida em seguida. A próxima versão do padrão foi originalmente chamada SQL3, mas atualmente é conhecida como SQL-99. Vamos tratar, prioritariamente, da última versão da SQL.

8.1 Definição de Dados e Tipos de Dados SQL 149

A SQL é uma linguagem de banco de dados abrangente: ela possui comandos para definição de dados, consultas e atualizações. Assim, ela tem ambas as DDL e DML. Além disso, tem funcionalidades para a definição de visões (*views*) no banco de dados, a fim de especificar a segurança e as autorizações para as definições de restrições de integridade e de controles de transação. Ela também possui regras para embutir os comandos SQL em linguagens de programação genérica como Java, COBOL ou C/C++.

Discutiremos a maioria desses tópicos nas próximas subseções.

Como a especificação do padrão SQL está em expansão, com mais funcionalidades a cada versão, o último padrão SQL-99 foi dividido em uma especificação de **núcleo** (*core*) mais **pacotes** (*packages*) opcionais. O núcleo deve ser implementado por todos os vendedores de SGBDs relacionais compatíveis com o padrão. Os pacotes podem ser implementados como módulos opcionais, que podem ser adquiridos independentemente para as aplicações específicas de um banco de dados, como *data mining* (garimpagem de dados), dados espaciais, dados temporais, *data warehousing*, processamento analítico on-line (OLAP), dados multimídia, e assim por diante. Faremos um resumo de alguns desses pacotes no final deste capítulo — e onde mais forem discutidos neste livro.

Como a SQL é muito importante (e muito grande), desenvolvemos dois capítulos para suas funcionalidades básicas. Neste capítulo, a Seção 8.1 descreve os comandos DDL da SQL para a criação de esquemas e tabelas e fornece uma visão geral dos tipos de dados básicos na SQL. A Seção 8.2 apresenta como são especificadas as restrições básicas, como a integridade de chave e referencial. A Seção 8.3 discute os comandos para a modificação de esquemas, tabelas e restrições. A Seção 8.4 descreve os construtores básicos da SQL para a especificação de resultados de consultas, e a Seção 8.5 discorre sobre as funcionalidades mais complexas das consultas SQL, como as funções agregadas e os agrupamentos. A Seção 8.6 descreve os comandos SQL para a inserção, exclusão e atualização de dados. A Seção 8.7 relaciona algumas funcionalidades SQL que serão apresentadas em outros capítulos do livro; essas funcionalidades incluem controle de transações no Capítulo 17, segurança/autorizações no Capítulo 23, banco de dados ativo (gatilhos — *triggers*) no Capítulo 24, funcionalidades de orientação a objetos no Capítulo 22, e funcionalidades OLAP (*OnLine Analytical Processing*— Processamento Analítico On-line) no Capítulo 28. A Seção 8.8 resume o capítulo.

No próximo capítulo, discutiremos os conceitos de visões (*views* — tabelas virtuais), então descreveremos como as restrições mais genéricas podem ser estabelecidas como asserções ou verificações. Em seguida, há uma descrição das diversas técnicas de programação para um banco de dados com SQL.

Para o leitor que desejar uma introdução menos abrangente da SQL, trechos da Seção 8.5 poderão ser pulados.

8.1 DEFINIÇÃO DE DADOS E TIPOS DE DADOS SQL

A SQL usa os termos **tabela**, **linha** e **coluna**, em vez dos termos *relação*, *tupla* e *atributo*, respectivamente, para o modelo relacional formal. Vamos usar os termos correspondentes de modo intercambiável. O principal comando SQL para a definição de dados é o CREATE, que pode ser usado para criar esquemas, tabelas (relações) e domínios [da mesma forma que outros construtores, como visões (*views*), asserções (*assertions*) e gatilhos (*triggers*)]. Antes de descrevermos o importante comando CREATE, vamos discutir os conceitos de esquema e catálogo na Seção 8.1.1 para colocar nossa discussão em perspectiva. A Seção 8.1.2 descreve como as tabelas são criadas, e a Seção 8.1.3 discute os tipos mais importantes disponíveis para a especificação de atributos. Como a especificação SQL é muito extensa, daremos uma descrição das funcionalidades mais importantes. Os detalhes adicionais poderão ser encontrados nos diversos documentos sobre o padrão SQL (notas bibliográficas).

8.1.1 Conceitos de Esquema e Catálogo em SQL

As primeiras versões da SQL não incluíam os conceitos de um esquema de um banco de dados relacional; todas as tabelas (relações) eram consideradas parte do mesmo esquema. O conceito de um esquema SQL foi incorporado a partir da SQL2, de modo a agrupar as tabelas e outros construtores que pertencem à mesma aplicação de um banco de dados. Um **esquema** SQL é identificado por um nome de esquema e inclui uma identificação de **autorização**, que indica o usuário ou a conta a qual o esquema pertence, bem como os **descritores** de *cada elemento* do esquema. Os elementos de esquema incluem tabela, restrições, visões (*views*), domínios e outros construtores (como concessão de autoridade) que descrevem o esquema. Um esquema é criado via um comando CREATE SCHEMA, que pode incluir todos os elementos de definição do esquema. Alternativamente, o esquema

1 Originalmente, a SQL teve comandos para a criação e exclusão de índices em arquivos que representam as relações, mas foram extintos do padrão SQL já há algum tempo.

pode ser definido por um nome e um identificador de autorização, e os elementos podem ser estabelecidos depois. Por exemplo, o comando a seguir cria um esquema chamado EMPRESA, pertencente a um usuário cuja identificação é JSMITH:

CREATE SCHEMA EMPRESA AUTHORIZATION JSMITH;

Em geral, nem todos os usuários estão autorizados a criar esquemas e elementos de esquemas. O privilégio para a criação de esquemas, tabelas e outros construtores deve ser explicitamente concedido (*granted*) para as contas de usuários relevantes, para o administrador do sistema ou para o DBA (administrador do banco de dados).

Além do conceito de esquema, a SQL2 usa o conceito de **catálogo** — uma coleção de esquemas em um ambiente SQL que recebe um nome. Um **ambiente SQL** é basicamente uma instalação de um SGBD relacional-padrão SQL em um sistema de computador. Um catálogo sempre contém um esquema especial chamado INFORMATION_SCHEMA, que proporciona as informações sobre todos os esquemas do catálogo e todos os descritores de seus elementos. As restrições de integridade, como a integridade referencial, podem ser definidas entre as relações somente se existirem em esquemas dentro do mesmo catálogo. Os esquemas dentro do mesmo catálogo também podem compartilhar certos elementos, como as definições de domínio.

8.1.2 O Comando CREATE TABLE em SQL

O comando **CREATE TABLE** é usado para especificar uma nova relação, dando-lhe um nome e especificando seus atributos e restrições iniciais. Os atributos são definidos primeiro e, a cada atributo, é dado um nome, um tipo para especificar o domínio de seus valores e alguma restrição de atributo, como NOTNULL (não pode ser vazio). As restrições de chave, a integridade de entidade e a integridade referencial podem ser especificadas dentro do comando CREATE TABLE depois que os atributos forem declarados, ou poderão ser adicionadas depois, usando-se o comando ALTER TABLE (Seção 8.3). A Figura 8.1 mostra um exemplo para a definição dos comandos SQL de criação do esquema do banco de dados relacional apresentado na Figura 5.7.

De maneira geral, o esquema SQL, no qual as relações serão declaradas, está implicitamente definido dentro do ambiente no qual os comandos CREATE TABLE serão executados. Alternativamente, podemos anexar de maneira explícita o nome do esquema ao nome da relação, separados por um ponto. Por exemplo, ao escrever

```
CREATE TABLE EMPRESA.EMPREGADO . . .
```

em vez de

```
CREATE TABLE EMPREGADO . . .
```

como na Figura 8.1, podemos, explicitamente (em vez de implicitamente), fazer com que a tabela EMPREGADO faça parte do esquema EMPRESA.

As relações declaradas por intermédio dos comandos CREATE TABLE são chamadas **tabelas básicas** (ou relações básicas); isso significa que uma relação e suas tuplas são realmente criadas e armazenadas como um arquivo pelo SGBD. As relações básicas são diferenciadas das **relações virtuais**, criadas pelo comando CREATE VIEW (Seção 9.2), que pode ou não corresponder a um arquivo físico real. Em SQL, os atributos de uma tabela básica são considerados *ordenados na sequência na qual são especificados* dentro do comando CREATE TABLE. Entretanto, as linhas (tuplas) não são consideradas ordenadas dentro da relação.

8.1.3 Tipos de Dados de Atributos e Domínios em SQL

Os **tipos de dados** básicos para os atributos incluem numéricos, cadeia de caracteres (*string*), cadeia de bits, booleanos, data e horário.

- **Numéricos** — tipos de dados numéricos que englobam os números inteiros de vários tamanhos (INTEIROS ou INT e SMALLINT) e os números ponto flutuante (reais) de várias precisões (FLOAT ou REAL e DOUBLE PRECISION). Os números formatados podem ser declarados pelo uso de DECIMAL (*i,j*) — ou DEC(*i,j*) ou NUMERIC(*i,j*) —, em que *i*, a *precisão*, é o número total de dígitos decimais, e *j*, a *escala*, refere-se ao número de dígitos depois do ponto decimal. O *default* para a escala é zero e, para a precisão, é definido na implementação.

2 A SQL também inclui o conceito de *cluster* (agrupamento) de catálogos dentro de um mesmo ambiente, mas isso não fica muito claro se muitos níveis de aninhamento forem exigidos na maioria das aplicações.

8.1 Definição de Dados e Tipos de Dados SQL

(a)

```
CREATE TABLE EMPREGADO
```

(FNOME	VARCHAR(15)	NOT NULL ,
MINICIAL	CHAR,	
LNAME	VARCHAR(15)	NOT NULL,
SSN	CHAR(9)	NOT NULL,
DATANASC	DATE	
ENDEREÇO	VARCHAR(30) ,	
SEXO	CHAR,	
SALÁRIO	DECIMAL(10,2),	
SUPERSSN	CHAR(9) ,	
DNO	INT	NOT NULL ,

```
PRIMARY KEY (SSN) ,
```

```
FOREIGN KEY (SUPERSSN) REFERENCES EMPREGADO(SSN) ,
```

```
FOREIGN KEY (DNO) REFERENCES DEPARTAMENTO(DNUM) ) ;
```

```
CREATE TABLE DEPARTAMENTO
```

(DNOME	VARCHAR(15)	NOT NULL ,
DNUMERO	INT	NOT NULL ,
GERSSN	CHAR(9)	NOT NULL ,
GERDATAINICIO	DATE,	

```
PRIMARY KEY (DNUM) ,
```

```

UNIQUE (DNOME) ,
FOREIGN KEY (MGRSSN) REFERENCES EMPREGADO(SSN) ) ;
CREATE TABLE DEPT_LOCALIZACOES
(DNUM          INT          NOT
DLOCACAO       VARCHAR(15)  NOT
PRIMARY KEY (DNUM, DLOCACAO) ,
FOREIGN KEY (DNUM) REFERENCES DEPARTAMENTO(DNUM) CREATE TABLE PROJETO
VARCHAR(15) INT
VARCHAR(15) , INT
NOT NOT
NULL, NULL,
);
NULL, NULL,
(PNOME
PNUMERO
PLOCALIZACAO
DNUM          INT          NOT NULL ,
PRIMARY KEY (PNUM) , UNIQUE (PNOME) ,
FOREIGN KEY (DNU) REFERENCES DEPARTAMENTO(DNUM)) ; CREATE TABLE TRABALHA_EM
( ESN          CHAR(9)      NOT NULL ,
PNO            INT          NOT NULL,
HORAS          DECIMAL(3,1) NOT NULL ,
PRIMARY KEY (ESSN, PNO) ,
FOREIGN KEY (ESSN) REFERENCES EMPREGADO(SSN) , FOREIGN KEY (PNO) REFERENCES
PROJETO(PNUM) ) ; CREATE TABLE DEPENDENTE
(ESSN          CHAR(9)      NOT NULL ,
DEPENDENTJMAME VARCHAR(15)  NOT NULL,
SEX            CHAR,
DATANASC       DATE,
PARENTESCO     VARCHAR(8) ,
PRIMARY KEY (ESSN, DEPENDENTE_NOME) ,
FOREIGN KEY (ESSN) REFERENCES EMPREGADO(SSN) ) ;

```

FIGURA 8.1 Comandos de definição de dados SQL CREATE TABLE para a definição do esquema EMPRESA da Figura 5.7

1 52 Capítulo 8 SQL-99: Definição de Esquema, Restrições Básicas e Consultas (*Queries*)

- **Cadeia de caracteres** — os tipos de dados cadeia de caracteres ou têm tamanho fixo — CHAR(n) ou CHARACTER(n), em que n é o número de caracteres — ou têm tamanho variável — VARCHAR(n) ou CHAR VARYING ou CHARACTER VARYING (n), em que n é o número máximo de caracteres. Quando se especifica um valor cadeia de caractere literal, este deve ser colocado entre os pares de aspas simples (apóstrofes) e é *case sensitive* (é feita a distinção entre as letras maiúsculas e minúsculas). Para as cadeias de tamanho fixo, uma cadeia menor é acoplada aos caracteres brancos à direita. Por exemplo, se o valor 'Smith' for conferido a um atributo tipo CHAR(10), ele terá acoplado cinco caracteres brancos para se tornar 'Smith' conforme necessário. Os brancos acoplados são normalmente ignorados quando as cadeias são comparadas. Para fins de comparação, as cadeias de caracteres são consideradas ordenadas alfabeticamente (ou lexicograficamente); se uma cadeia str1 aparece antes de uma cadeia str2 em ordem alfabética, então o str1 é considerado menor que o str2. Existe também a concatenação de operadores denotada por || (barras verticais duplas), que podem concatenar duas cadeias em SQL. Por exemplo, 'abe' || 'XYZ' resulta na cadeia única 'abcXYZ'.
- **Bit-string** (cadeia de bits) — são os tipos de dados *bit-string* ou de tamanho fixo n — BIT(n) — ou têm tamanho variável — BIT VARYING(n) — em que n é o número máximo de bits. O *default* para n, o tamanho para uma cadeia de caracteres ou *bit-string*, é 1. Os *bit-strings* literais são colocados entre aspas simples, mas precedidos por B para distingui-los das cadeias de caracteres; por exemplo, B'10101'.
- **Booleano** — um tipo de dados booleano possui os valores tradicionais de TRUE (verdadeiro) ou FALSE (falso). Em SQL, em virtude da presença dos valores NULL, são usados três valores lógicos, assim, um terceiro valor possível para o tipo de dado booleano é UNKNOWN (desconhecido). Discutiremos a necessidade dos valores UNKNOWN e esse terceiro valor lógico na Seção 8.5.1.
- **Date e time** — os novos tipos de dados *date* (data) e *time* (horário) foram acrescentados à SQL2. O tipo de dado DATE contém dez posições e seus componentes são YEAR (ano), MONTH (mês) e DAY (dia) no formato YYYY-MM-DD. O tipo de dado TIME tem, no mínimo, oito posições, com os componentes HOUR (hora), MINUTE (minuto) e SECOND (segundo) no formato HH:MM:SS. Somente datas e horários nos formatos permitidos podem ser implementados na SQL. A comparação < (menor que) pode ser usada com datas ou horários — uma data *anterior* é considerada menor que uma data posterior, e de modo similar são tratados os horários. Os valores literais são representados por cadeias de caracteres entre aspas simples precedidas das palavras-chave DATE ou TIME; por exemplo, DATE '2002-09-27' ou TIME '09:12:47'. Além disso, um tipo de dado TIME(i), em que i é chamado *precisão de tempo em fração de segundos*, especifica i + 1 posições adicionais para TIME — uma posição para um caractere separador adicional e i posições para a especificação de frações decimais de um segundo. Um tipo de dado TIME WITH TIME ZONE insere seis posições adicionais para a *especificação* do *default* universal zona de tempo, que compreende o intervalo +13:00 a -12:59 em unidades de HOURS:MINUTES. Se não for incluído o WITH TIME ZONE, o *default* é o horário local para a sessão SQL.
- **Timestamp** — um tipo de dados TIMESTAMP engloba os campos DATE e TIME, mais um mínimo de seis posições para frações decimais de segundos e uma qualificação opcional WITH TIME ZONE. Os valores literais são representados por cadeias entre aspas simples precedidas da palavra-chave TIMESTAMP, com um espaço em branco entre data e horário; por exemplo, TIMESTAMP '2002-09-27 09:12:47 648302'.
- **Interval** — outro tipo de dado relacionado a DATE, TIME e TIMESTAMP é o tipo de dado INTERVAL, que especifica um **intervalo** — um *valor relativo* que pode ser utilizado para incrementar ou decrementar um valor absoluto de data, horário ou

timestamp. Os valores *intervals* são qualificados para ser intervalos YEAR/MONTH (ano/mês) ou para intervalos DA Y/TIME (dia/hora).

- Os formatos DATE, TIME e TIMESTAMP podem ser considerados um tipo especial de cadeias de caracteres. Assim, podem, de modo geral, ser usados em comparações quando **lançados (forçados)** ou convertidos) em cadeias equivalentes.

É possível especificar o tipo de dado de cada atributo diretamente, como na Figura 8.1, porém, um domínio pode ser declarado e o nome do domínio usado com a especificação do atributo. Isso torna mais fácil a troca do tipo de dado para um domínio, que é usado por muitos atributos em um esquema, além de melhorar a legibilidade do esquema. Por exemplo, podemos criar um domínio TIPO_SSN por meio do seguinte comando:

CREATE DOMAIN TIPO_SSN AS CHAR(9);

3 Esse não é o caso das palavras-chave SQL, como CREATE ou CHAR. As palavras-chave SQL não são *case sensitive*; a SQL trata as maiúsculas e as minúsculas de modo equivalente.

4 Para os caracteres não-alfabéticos existe uma ordem definida.

5 O *bit strings*, cujo tamanho é um múltiplo de quatro, também pode ser especificado em notação *hexadecimal*, com a cadeia literal precedida por X e cada caractere hexadecimal representando quatro bits.

8.2 Especificando as Restrições Básicas em SQL

153

Podemos usar `TIP0_SSN` no lugar de `CHAR(9)`, na Figura 8.1, para os atributos `SSN` e `SUPERSSN` de `EMPREGADO`, `GERSSN` de `DEPARTAMENTO`, `ESSN` de `TRABALHA_EM`, e `ESSN` de `DEPENDENTE`. Um domínio também pode ter uma especificação *default* opcional por meio da cláusula `DEFAULT`, como discutiremos, mais tarde, em atributos.

8.2 ESPECIFICANDO AS RESTRIÇÕES BÁSICAS EM SQL

Discutiremos agora as restrições básicas que podem ser definidas na SQL como parte da criação de uma tabela. Elas englobam as restrições de chave e referencial, bem como as restrições de domínio de atributo e `NULLs`, e as restrições para as tuplas individuais dentro de uma relação. Apresentaremos a especificação de restrições mais gerais, chamadas asserções, na Seção 9.1.

8.2.1 Especificando as Restrições de Atributo e Padrões (*default*) de Atributos

Como a SQL permite `NULLs` como valores de atributos, uma *restrição NOTNULL* pode ser especificada se o `NULL` não for permitido para um dado atributo. Essa restrição está sempre implícita para os atributos que são designados como *chaves primárias* de cada relação, mas pode ser especificada para qualquer outro atributo cujos valores exigirem preenchimento diferente de `NULL`, como mostrado na Figura 8.1.

É também possível definir um *valor default* para um atributo por meio da adição da cláusula `DEFAULT <valor>` na definição de um atributo. O valor *default* será inserido em qualquer nova tupla, caso não seja proporcionado nenhum valor explícito para esse atributo. A Figura 8.2 ilustra um exemplo para a especificação de um gerente *default* para um novo departamento, e de um departamento *default* para um novo empregado. Se não for especificada nenhuma cláusula *default*, o *valor default* será `NULL` para os atributos *que não tiverem* uma restrição `NOT NULL`.

Outro tipo de restrição pode limitar os valores do atributo ou de seu domínio pelo uso da cláusula `CHECK`, seguida da definição do atributo ou do domínio. Por exemplo, suponha que os números do departamento estejam restritos aos números inteiros entre 1 e 20; então podemos alterar a declaração do atributo `DNUMERO` na tabela `DEPARTAMENTO` (Figura 8.1) conforme segue:

```
DNUMERO INT NOT NULL CHECK (DNUMERO > 0 AND DNUMERO < 21);
```

A cláusula `CHECK` também pode ser usada em conjunto com o comando `CREATEDOMAIN`. Por exemplo, podemos escrever o seguinte comando:

```
CREATE DOMAIN D_NUM AS INTEGER CHECK (D_NUM > 0 AND D_NUM < 21);
```

Podemos usar, então, o domínio criado `D_NUM` como o tipo atributo para todos os atributos que se referirem aos números do departamento da Figura 8.1, como `DNUMERO` de `DEPARTAMENTO`, `DNUM` de `PROJETO`, `DNO` de `EMPREGADO`, e assim por diante.

8.2.2 Especificando as Restrições de Chave e de Integridade Referencial

Como as restrições de chave e de integridade referencial são muito importantes, existem cláusulas especiais dentro do comando `CREATE TABLE` para especificá-las. A Figura 8.1 mostra alguns exemplos para ilustrar a especificação das chaves e da integridade referencial.

A cláusula `PRIMARY KEY` especifica um ou mais atributos que definem a chave primária da relação. Se a chave primária tiver um atributo *único*, a cláusula pode seguir o atributo diretamente. Por exemplo, a chave primária de `DEPARTAMENTO` poderia ser especificada como segue (como alternativa ao modo especificado na Figura 8.1):

```
DNUMERO INT PRIMARY KEY;
```

A cláusula `UNIQUE` define as chaves alternativas (secundárias), como ilustrado na declaração das tabelas `DEPARTAMENTO` e `PROJETO` na Figura 8.1.

A integridade referencial é especificada pela cláusula `FOREIGN KEY` (chave estrangeira), como mostra a Figura 8.1. Como discutimos na Seção 5.2.4, uma restrição de integridade referencial pode ser violada quando as tuplas são inseridas ou

6 A cláusula `CHECK` também pode ser usada para outras finalidades, conforme veremos.

7 As restrições de chave e de integridade referencial não existiam nas primeiras versões da SQL. Em algumas implementações mais antigas, as chaves eram especificadas implicitamente, no nível interno, via comando `CREATE INDEX`.

154 Capítulo 8 SQL-99: Definição de Esquema, Restrições Básicas e Consultas (*Queries*)

deletadas, ou quando os valores dos atributos referentes à chave estrangeira ou à chave primária forem modificados. A ação-padrão da SQL, quando há uma violação de integridade, é **rejeitar** a operação de atualização que iria causar a violação. No entanto, o projetista do esquema pode especificar uma ação alternativa, caso uma restrição de integridade seja violada, por meio da anexação de uma cláusula ação **referencial engatilhada** (*referential triggered action*) em uma restrição de chave estrangeira. As opções incluem SET NULL, CASCADE e SET DEFAULT. Uma dessas opções deve ser escolhida com ON DELETE ou ON UPDATE. Na Figura 8.2, ilustramos esse procedimento nos exemplos apresentados. Ali, o projetista do banco de dados optou por SET NULL ON DELETE (na exclusão, marcar nulo) e CASCADE ON UPDATE (na atualização, propagar) para as chaves estrangeiras SUPERSSN de EMPREGADO. Isso significa que, se uma tupla de um empregado supervisor for *deletada*, o valor para SUPERSSN será automaticamente marcado com NULL em todas as tuplas de empregados que fizerem referência à tupla supervisor que foi excluída. Porém, se um valor SSN para um empregado supervisor for *atualizada* (digamos, porque esse valor foi registrado incorretamente), o novo valor é *propagado* (*cascaded*) para o SUPERSSN de todas as tuplas de empregados que fizerem referência à tupla empregado atualizada.

```
CREATE TABLE EMPREGADO (...,
DNO INT NOTNULL DEFAULT 1,
CONSTRAINT EMPPK
PRIMARY KEY (SSN) , CONSTRAINT EMPSUPERFK FOREIGN KEY (SUPERSSN) REFERENCES
EMPREGADO(SSN) ON DELETE SET NULL ON UPDATE CASCADE , CONSTRAINT EMPDEPTFK
FOREIGN KEY (DNO) REFERENCES DEPARTAMENTO(DNUMERO) ON DELETE SET DEFAULT ON
UPDATE CASCADE );
CREATE TABLE DEPARTAMENTO (...,
GERSSN CHAR(9) NOT NULL DEFAULT '888665555',
CONSTRAINT DEPTPK
PRIMARY KEY (DNUMERO) , CONSTRAINT DEPTSK
UNIQUE (DNOME), CONSTRAINT DEPTMGRFK
FOREIGN KEY (GERSSN) REFERENCES EMPREGADO(SSN) ON DELETE SET DEFAULT ON UPDATE
CASCADE );
CREATE TABLE DEPJ.OCALIZACOES (...,
PRIMARY KEY (DNUMERO, DLOCALIZACAO),
FOREIGN KEY (DNUMERO) REFERENCES DEPARTAMENTO(DNUMERO)
ON DELETE CASCADE ON UPDATE CASCADE ) ;
```

FIGURA 8.2 Exemplo ilustrando como os valores do atributo *default* e as ações referenciais engatilhadas são especificados em SQL.

Em geral, a ação do SGBD para SET NULL ou SET DEFAULT é a mesma, tanto para ON DELETE quanto para ON UPDATE: o valor que afeta o valor do atributo referido receberá o valor NULL para SET NULL, ou receberá o valor definido como *default* para SET DEFAULT. A ação para CASCADE ON DELETE é excluir todas as tuplas referidas, enquanto a ação para CASCADE ON UPDATE é trocar os valores da chave estrangeira de todas as tuplas pelo valor atualizado (novo) da chave primária à qual faz referência. É responsabilidade do projetista do banco de dados escolher a ação apropriada para a especificação no esquema do banco de dados. Como regra, a opção CASCADE é apropriada para as relações de 'relacionamento' (Seção 7.1), como TRABALHA_EM; para relações que representam atributos multivalorados, como DEP_LOCALIZACOES, e para relações que representam tipos entidades fracas, como DEPENDENTE.

8.3 Comandos para as Alterações de Esquemas SQL

155

8.2.3 Denominando as Restrições

A Figura 8.2 também ilustra como uma restrição pode ser **nomeada** após a palavra CONSTRAINT. O nome de cada uma das restrições de um esquema em particular deve ser único. O nome da restrição é usado para identificar uma restrição em particular, caso essa restrição tenha de ser eliminada posteriormente e redefinida como outra restrição, como discutiremos na Seção 8.3. Nomear uma restrição é opcional.

8.2.4 Especificando as Restrições em Tuplas Usando CHECK (Checar)

Além das restrições de chave e referencial, que são especificadas por palavras-chave especiais, outras restrições podem ser definidas pela cláusula adicional CHECK no final da declaração CREATETABLE. Podemos chamar esse procedimento de restrição com **base em tupla**, pois se aplica a cada tupla, *individualmente*, e será verificada sempre que uma tupla for inserida ou modificada. Por exemplo, suponha que a tabela DEPARTAMENTO, da Figura 8.1, tenha um atributo adicional DEP_CRIA_DATA que armazena a data de criação do departamento. Poderíamos adicionar a cláusula CHECK, no final da declaração CREATE TABLE, da tabela DEPARTAMENTO para ter certeza de que a data da nomeação de seu gerente é posterior à data de criação do departamento:

```
CHECK (DEP_CRIA_DATA < GERDATAINICIO);
```

Na declaração CREATE ASSERTION da SQL, a cláusula CHECK pode ser usada também para especificar as restrições mais genéricas. Discutiremos esse assunto na Seção 9.1, dado que, para isso, é necessário utilizar todo o potencial das consultas (*queries*), objeto de discussão das seções 8.4 e 8.5.

8.3 COMANDOS PARA AS ALTERAÇÕES DE ESQUEMAS SQL

Nesta seção daremos uma visão geral dos **comandos para evolução de um esquema**, disponíveis em SQL, que podem ser usados para alterar um esquema por intermédio da adição ou da eliminação de tabelas, atributos, restrições e outros elementos de esquema.

8.3.1 O Comando DROP (eliminar)

O comando DROP pode ser usado para eliminar os elementos de esquemas *nomeados*, como tabelas, domínios ou restrições. Pode, também, eliminar o esquema propriamente dito. Por exemplo, se um esquema não for mais necessário, o comando DROP SCHEMA pode ser utilizado. Há duas opções de *comportamento* para o comando *drop*: CASCADE e RESTRICT. Por exemplo, para remover o esquema EMPRESA com todas as suas tabelas, domínios e outros elementos, a opção CASCADE deve ser usada conforme segue:

```
DROP SCHEMA EMPRESA CASCADE;
```

Se for usada a opção RESTRICT em vez de CASCADE, o esquema é eliminado somente se não contiver *nenhum elemento*, caso contrário, o comando DROP não será executado.

Se uma relação básica não é mais necessária em um esquema, essa relação e suas especificações poderão ser eliminadas por meio do comando DROP TABLE. Por exemplo, se não desejarmos mais manter os dados dos dependentes dos empregados no banco de dados EMPRESA da Figura 8.1, poderíamos nos livrar da relação DEPENDENTE emitindo o seguinte comando:

```
DROP TABLE DEPENDENTE CASCADE;
```

Se for escolhida a opção RESTRICT em vez de CASCADE, a tabela será eliminada somente se *não for referência* em nenhuma restrição (por exemplo, em definições de chaves estrangeiras de outras relações) ou visões (*views* — Seção 9.2). Com a opção CASCADE, todas as restrições e as visões que fizerem referência à tabela serão automaticamente eliminadas do esquema, bem como a tabela propriamente dita.

O comando DROP também pode ser usado para eliminar outros tipos de elementos que possuam nomes, como restrições e domínios.

8.3.2 O Comando ALTER

As definições de uma tabela básica ou de outros elementos do esquema que possuírem denominação poderão ser alteradas pelo comando ALTER. Para as tabelas básicas, as *ações de alteração* possíveis compreendem: adicionar ou eliminar uma coluna (atributo), alterar a definição de uma coluna e adicionar ou eliminar restrições na tabela. Por exemplo, para adicionar um atributo de modo a armazenar as funções exercidas pelos empregados na relação básica EMPREGADO do esquema EMPRESA, podemos usar o comando:

```
ALTER TABLE EMPRESA.EMPREGADO ADD FUNCAO VARCHAR ( 12 ) ;
```

Deveríamos, então, entrar com um valor para o novo atributo FUNCAO de cada tupla de EMPREGADO. Isso pode ser feito pela especificação da cláusula *default* ou por meio do comando UPDATE (Seção 8.6). Se não for especificada nenhuma cláusula *default*, o novo atributo conterá NULLs, imediatamente após a execução do comando, em todas as tuplas da relação; logo, a restrição NOT NULL não é permitida nesse caso.

Para eliminar uma coluna, devemos optar por CASCADE ou RESTRICT em termos de comportamento para a eliminação. Se for escolhida a opção CASCADE, todas as restrições e as visões que fizerem referência a essa coluna serão automaticamente eliminadas do esquema, bem como a coluna em questão. Se for escolhida RESTRICT, o comando será bem-sucedido somente se nenhuma visão ou restrição (ou outros elementos) fizerem referência à coluna. Por exemplo, o seguinte comando remove o atributo ENDEREÇO da tabela básica EMPREGADO:

```
ALTER TABLE EMPRESA.EMPREGADO DROP ENDEREÇO CASCADE;
```

Também é possível alterar uma definição de coluna eliminando uma cláusula *default* existente ou definindo uma nova cláusula *default*. Os exemplos a seguir ilustram essas cláusulas:

```
ALTER TABLE EMPRESA.EMPREGADO ALTER GERSSN DROP DEFAULT;
```

```
ALTER TABLE EMPRESA.EMPREGADO ALTER GERSSN SET DEFAULT  
" 33344555 " ;
```

Podem ser modificadas, ainda, as restrições especificadas em uma tabela adicionando ou eliminando uma restrição. Para ser eliminada, uma restrição precisa ter recebido um nome quando foi especificada. Por exemplo, para eliminar a restrição denominada EMPSUPERFK, da Figura 8.2, especificada na relação EMPREGADO, escrevemos:

```
ALTER TABLE EMPRESA.EMPREGADO DROP CONSTRAINT EMPSUPERFK  
CASCADE;
```

Uma vez feito isso, poderemos redefinir, se necessário, uma restrição substituta adicionando-a à relação. Isso pode ser feito pela palavra-chave ADD, no comando ALTER TABLE, seguida da nova restrição, que pode, ou não, receber um nome, podendo ser qualquer um dos tipos de restrições de tabelas já discutidos.

As subseções precedentes deram uma visão geral dos comandos para a evolução de esquema na SQL. Existem muitos outros detalhes e opções, e sugerimos ao leitor interessado uma lista de documentos SQL nas notas bibliográficas. As duas próximas seções discutem as potencialidades das consultas SQL.

8.4 CONSULTAS SQL BÁSICAS

A SQL possui um comando básico para a recuperação de informações de um banco de dados: o comando SELECT. O comando SELECT não tem nenhuma relação com a operação SELECT da álgebra relacional, que foi discutida no Capítulo 6. Existem diversas opções e variações do comando SELECT na SQL, assim, vamos introduzindo gradualmente suas funcionalidades. Usaremos o esquema da Figura 5.5 para os exemplos de consultas, e vamos fazer referência ao exemplo de estado desse banco de dados, apresentado na Figura 5.6, para mostrar os resultados de alguns dos exemplos de consultas.

Antes de prosseguir, precisamos expor uma importante distinção entre a SQL e o modelo relacional formal discutido no Capítulo 5: a SQL permite que uma tabela (relação) tenha duas ou mais tuplas idênticas em todos os valores de seus atributos. Logo, em geral, uma tabela SQL não é um *conjunto de tuplas* porque esse conjunto não permitiria dois membros idênticos; isso seria um multiconjunto (*multiset*, algumas vezes chamado *bag*) de tuplas. Algumas relações SQL são *restritas a conjuntos*, caso uma restrição de chave tenha sido declarada ou uma opção DISTINCT tenha sido usada no comando SELECT (descrito no final desta seção). Devemos estar cientes dessa distinção, como discutiremos nos exemplos.

8.4.1 A Estrutura SELECT-FROM-WHERE das Consultas Básicas

As consultas em SQL podem ser muito complexas. Começaremos com as consultas mais simples e, progressivamente, chegaremos às mais complexas, passo a passo. O formato básico da declaração SELECT, algumas vezes chamada **mapeamento** ou **bloco select-from-where**, é composto por três cláusulas: SELECT, FROM e WHERE, e tem a seguinte forma:

SELECT <lista de atributos> **FROM** <lista de tabelas> **WHERE** <condicao>;

em que

- <lista de atributos> é uma lista dos nomes dos atributos cujos valores serão recuperados pela consulta.
- <lista de tabelas> é uma lista dos nomes das relações necessárias para o processamento da consulta.
- <condicao> é uma expressão condicional (booleana) que identifica as tuplas que serão recuperadas pela consulta.

Em SQL, os operadores lógicos básicos de comparação usados para confrontar os valores entre os atributos e constantes são: =, <, <=, >, >= e <>. Esses operadores correspondem aos da álgebra relacional =, <, <=, >, >= e ^, respectivamente; e com os operadores =, <, <=, >, >= e != da linguagem de programação C/C++. A principal diferença está no operador da desigualdade (*not equal*). Em SQL há diversos operadores de comparação adicionais que vamos apresentar gradualmente, conforme necessário.

Ilustraremos agora o comando básico SELECT em SQL com alguns exemplos de consultas. As consultas foram rotuladas, aqui, com o mesmo número com que aparecem no Capítulo 6, para facilitar o cruzamento de referências.

Consulta 0

Recupere o aniversário e o endereço do(s) empregado(s) cujo nome seja 'John B. Smith'.

QO: SELECT DATANASC, ENDEREÇO FROM EMPREGADO WHERE PNOME='John' AND MINICIAL='B' AND UNOME='Smith';

Essa consulta envolve somente a relação EMPREGADO relacionada na cláusula FROM. A consulta *seleciona* as tuplas de EMPREGADO que satisfazem a condição da cláusula WHERE, então *projeta* o resultado dos atributos DATANASC e ENDEREÇO relacionados na cláusula SELECT. QO é similar à seguinte expressão da álgebra relacional, exceto pelas repetições (de tuplas) que, se houver, poderiam *não* ser eliminadas:

'''DATANASC, ENDEREÇO(>NOME= 'John' AND MINICIAL='B' AND UNOME='Smi th' (EMPREGADO))

Logo, uma consulta SQL, com apenas um nome de relação na cláusula FROM, é similar a um par de operações SELECT-PROJECT da álgebra relacional. A cláusula SELECT da SQL especifica os *atributos para a projeção*, e a cláusula WHERE especifica a *condição de seleção*. A única diferença é que, em uma consulta SQL, podemos obter tuplas repetidas no resultado, uma vez que a restrição de que uma relação seja um conjunto não é incorporada. A Figura 8.3a mostra o resultado da consulta QO no banco de dados da Figura 5.6.

A consulta QO é similar também à seguinte expressão do cálculo relacional de tupla, exceto pelas repetições que, se houver, poderiam *não* ser eliminadas em uma consulta SQL:

QO: {t.DATANASC, t.ENDEREÇO | EMPREGADO(t) AND t.PNOME='John' AND t.MINICIAL='B' AND t.UNOME='Smith'}

Assim, podemos pensar em uma variável de tupla implícita, dentro de um intervalo da consulta SQL, sobre cada tupla da tabela EMPREGADO, e evoluir a condição para a cláusula WHERE. Somente aquelas tuplas que satisfizerem a condição — isto é, aquelas para as quais a condição for VERDADEIRA, depois de substituídos os valores correspondentes de seus atributos — serão selecionadas.

Consulta 1

Recupere o nome e o endereço de todos os empregados que trabalham no departamento 'Pesquisa'.

Q1: SELECT PNOME, UNOME, ENDEREÇO FROM EMPREGADO, DEPARTAMENTO WHERE DNAME='Pesquisa' AND DNUMERO=DNO;

158

Capítulo 8 SQL-99: Definição de Esquema, Restrições Básicas e Consultas (*Queries*)

A Consulta Q1 é similar à sequência das operações SELECT-PROJECT-JOIN da álgebra relacional. Essas consultas são frequentemente chamadas **consultas select-project-join**. Na cláusula WHERE de Q1, a condição DNOME = 'Pesquisa' é a condição de seleção e corresponde à operação SELECT da álgebra relacional. A condição DNUMERO = DNO é a **condição de junção (join condition)**, que corresponde à condição de junção (JOIN) da álgebra relacional. O resultado da consulta Q1 é apresentado na Figura 8.3b. Em geral, pode ser especificado qualquer número de condições de seleção e junção em uma única consulta SQL. O próximo exemplo é uma consulta *select-project-join* com *duas* condições de junção.

Consulta 2

Para cada projeto localizado em 'Stafford', relacione o número do projeto, o número do departamento responsável e o último nome do gerente do departamento, seu endereço e sua data de aniversário.

Q2: SELECT PNUMERO, DNUM, UNOME, ENDEREÇO, DATANASC FROM PROJETO, DEPARTAMENTO, EMPREGADO WHERE DNUM=DNUMERO AND GERSSN=SSN AND PLOCALIZACAO='Stafford';

A condição de junção DNUM = DNUMERO relaciona um projeto a seu departamento de controle, enquanto a condição de junção GERSSN = SSN relaciona o departamento de controle com o empregado que administra esse departamento. A Figura 8.3c apresenta o resultado da consulta Q2.

(a) **DATANASC**

1965-01-09

ENDEREÇO

731 Fondren, Houston, TX

(b) **PNUMO LNAME**

John Smith
Franklin Wong
Ramesh Narayan
Joyce English

ENDEREÇO

731 Fondren, Houston, TX 638 Voss, Houston, TX 975 Fire Oak, Humble, TX 5631 Rice, Houston, TX

(c) **PNUMERO DNUM LNAME ENDEREÇO DATANASC**

10 30

Wallace Wallace

291 Berry, Bellaire, TX 291 Berry, Bellaire, TX

1941-06-20 1941-06-20

(d) E.PNUMO E.UNOME S.PNUMO S.UNOME

(f) **SSN**

DNOME

(e)

(g)

John	Smith	Franklin	Wong	123456789	Pesquisa
Franklin	Wong	James	Borg	333445555	Pesquisa
Alicia	Zelaya	Jennifer	Wallace	999887777	Pesquisa
Jennifer	Wallace	James	Borg	987654321	Pesquisa
Ramesh	Narayan	Franklin	Wong	666884444	Pesquisa
Joyce	English	Franklin	Wong	453453453	Pesquisa
Ahmad	Jabbar	Jennifer	Wallace	987987987 888665555 123456789	Pesquisa Administração
SSN				333445555 999887777	Administração Administração
123456789				987654321	Administração
333445555				666884444	Administração
999887777				453453453	Administração
987654321				987987987	Administração
666884444				888665555	Administração
453453453				123456789	Sede Administrativa
987987987				333445555	Sede Administrativa
888665555				999887777 987654321 666884444 453453453 987987987 888665555	Sede Administrativa Sede Administrativa Sede Administrativa Sede Administrativa Sede

PNUMO MINICIAL	UNOME	SSN	DATANASC	ENDEREÇO	SEXO	SALÁRIO	SUPERSSN	DN
John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000	333445555 5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555 5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555 5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555 5

FIGURA 8.3 Resultado das consultas SQL, quando aplicadas ao banco de dados EMPRESA, no estado, mostrado na Figura 5.6. (a) Q0. (b) Q1. (c) Q2. (d) Q8. (e) Q9. (f) Q10. (g) Q1C.

8.4.2 Nomes de Atributos Ambíguos, *Aliases* (Pseudônimos) e Variáveis de Tuplas

Em SQL, um mesmo nome pode ser usado para dois (ou mais) atributos desde que esses atributos estejam em *relações diferentes*. Se esse é o caso e se uma consulta se refere a dois ou mais atributos com o mesmo nome, é preciso **qualificar** o nome do atributo com o nome da relação, de modo a prevenir ambigüidade. Isso é feito por meio da *prefixação* do nome da relação ao nome do atributo, separados por um ponto. Para ilustrar, suponha que os atributos DNO e UNOME da relação EMPREGADO, nas figuras 5.5 e 5.6, fossem chamados DNUMERO e NOME, e que o atributo DNO de DEPARTAMENTO também fosse chamado NOME; então, para prevenir ambigüidade, a consulta Q1 deveria ser refeita, como mostrado na Q1 A. Precisaríamos prefixar os atributos NOME e DNUMERO em Q1 A para especificar a qual deles estamos fazendo referência, uma vez que os mesmos nomes de atributos são usados em ambas as relações:

Q1A: SELECT P.NOME, EMPREGADO.NOME, ENDEREÇO FROM EMPREGADO, DEPARTAMENTO WHERE DEPARTAMENTO.NOME='Pesquisa' AND DEPARTAMENTO.DNUMERO=EMPREGADO.DNUMERO;

A ambigüidade também pode ser originada no caso de consultas que se referem duas vezes à mesma relação, como no exemplo seguinte.

Consulta 8

Para cada empregado, recupere o primeiro e o último nome do empregado e o primeiro e o último nome de seu superior imediato.

Q8: SELECT E.PNOME, E.UNOME, S.PNOME, S.UNOME FROM EMPREGADO AS E, EMPREGADO AS S WHERE E.SUPERSSN=S.SSN;

Nesse caso, podemos declarar nomes alternativos E e S, chamados *aliases* (pseudônimos) ou *variáveis de tuplas*, para a relação EMPREGADO. Um *alias* pode seguir a palavra-chave AS, como mostrado em Q8, ou pode seguir diretamente o nome da relação — por exemplo, escrevendo EMPREGADO e, EMPREGADO S na cláusula FROM de Q8. Também é possível usar pseudônimos para os atributos da relação dentro de uma consulta SQL por intermédio de *aliases*. Por exemplo, se escrevermos

EMPREGADO AS E(PN, MI, UN, SSN, DT, END, SEX, SAL, SSSN, DNO)

> na cláusula FROM, PN se torna pseudônimo de P.NOME, MI de MINICIAL, UN de UNOME, e assim por diante.

Em Q8 podemos imaginar que E e S são duas *cópias diferentes* da relação EMPREGADO; a primeira, E, representa os empregados no papel de supervisionados; a segunda, S, representa os empregados no papel de supervisores. Podemos, agora, juntar as duas cópias. Naturalmente, na verdade há *apenas uma* relação EMPREGADO, e a condição de junção compõe a relação com ela mesma por meio da paridade, entre as tuplas, na condição de junção E.SUPERSSN=S.SSN. Note que esse é um exemplo de consulta com recursividade nível um, conforme discutimos na Seção 6.4.2. Nas primeiras versões da SQL, bem como na álgebra relacional, não era possível especificar, em um único comando SQL, uma consulta recursiva geral com um número desconhecido de níveis. Foi incorporado um construtor para a especificação de consultas recursivas na SQL-99, conforme descrito no Capítulo 22.

A Figura 8.3d mostra o resultado da Q8. Sempre que houver em uma relação de um ou mais *aliases* especificados, poderemos usar esses nomes para representar diferentes referências para essa relação. Isso permite referências múltiplas para uma mesma relação dentro de uma consulta. Observe que, se quisermos, poderemos usar esse mecanismo de atribuição de pseudônimos (*alias*) em qualquer consulta SQL, na especificação de variáveis de tuplas para qualquer tabela na cláusula WHERE, precisando ou não fazer referência mais de uma vez à mesma relação. De fato, essa prática é recomendada, uma vez que acaba por

facilitar a compreensão de uma consulta. Por exemplo, poderíamos formular a consulta Q1A como em Q1B:

Q1B: SELECT E.PNOME, E.NOME, E.ENDEREÇO FROM EMPREGADO E, DEPARTAMENTO D WHERE D.NOME='Pesquisa' AND D.DNUMERO=E.DNUMERO;

Se especificarmos as variáveis de tupla para todas as tabelas na cláusula WHERE, uma consulta *select-project-join* em SQL reconstruirá, de forma muito parecida, a expressão correspondente no cálculo relacional de tupla (exceto pela eliminação das repetições). Por exemplo, compare Q1B com a seguinte expressão de cálculo relacional de tupla:

Q1: {e.PNOME, e.UNOME, e.ENDEREÇO I EMPREGADO (e) AND (3d)}

(DEPARTAMENTO(d) AND d.DNOME='PesquisA' AND d.DNUMERO=e.DN0) }

Observe que a principal diferença — além da sintaxe — é que na consulta SQL, o quantificador de existência não é apresentado explicitamente.

8.4.3 Cláusula WHERE Ausente e Uso do Asterisco

Discutiremos aqui outras duas funcionalidades da SQL. A *ausência* da cláusula WHERE indica que não há nenhuma condição para seleção de tuplas; logo, *todas as tuplas* da relação especificada na cláusula FROM estão qualificadas e serão selecionadas no resultado da consulta. Se for especificada mais de uma relação na cláusula FROM, e não existir cláusula WHERE, então será obtido o PRODUTO CARTESIANO dessas relações — *todas as possíveis combinações de tuplas* serão selecionadas. Por exemplo, a Consulta Q9 seleciona todos os SSNs de EMPREGADO (Figura 8.3e), e a Consulta Q10 seleciona todas as combinações de um SSN de EMPREGADO e um DNOME de DEPARTAMENTO (Figura 8.3f).

Consultas 9 e 10

Selecione todos os SSNs de EMPREGADO (Q9) e todas as combinações dos SSN de EMPREGADO e dos DNOME de DEPARTAMENTO (Q10) do banco de dados.

Q9: **SELECT** SSN

FROM EMPREGADO ;

Q10: **SELECT** SSN, DNOME

FROM EMPREGADO, DEPARTAMENTO;

É extremamente importante especificar todas as condições de seleção e de junção na cláusula WHERE; se alguma condição for omitida, podem ocorrer resultados incorretos ou muito grandes. Note que Q10 é similar à operação de PRODUTOCAR-TESIANO, seguida da operação PROJECT na álgebra relacional. Se especificarmos todos os atributos de EMPREGADO e DEPARTAMENTO em Q10, teríamos o PRODUTO CARTESIANO (exceto pela eliminação de repetições, se houver).

Para recuperar todos os valores dos atributos das tuplas selecionadas, não precisamos listar explicitamente todos os seus nomes na SQL; podemos especificar apenas um *asterisco* (*), que significa selecionar *todos os atributos*. Por exemplo, a consulta QIC recupera todos os valores dos atributos de EMPREGADO que trabalham no DEPARTAMENTO de número 5 (Figura 8.3g); a consulta Q1D recupera, para todo EMPREGADO do departamento 'Pesquisa', todos os atributos do EMPREGADO e todos os atributos do DEPARTAMENTO no qual eles trabalham; a consulta Q10A especifica o PRODUTO CARTESIANO das relações EMPREGADO e DEPARTAMENTO.

QIC: **SELECT** *

FROM EMPREGADO **WHERE** DNO=5;

Q1D: **SELECT** *

FROM EMPREGADO, DEPARTAMENTO **WHERE** DNOME='Pesquisa' AND DNO=DNUMERO;

Q10A: **SELECT**

FROM EMPREGADO, DEPARTAMENTO;

8.4.4 Tabelas como Conjuntos em SQL

Como havíamos mencionado anteriormente, a SQL geralmente trata uma tabela não como um conjunto, mas como um **multiconjunto**; *tuplas repetidas podem aparecer mais de uma vez* em uma tabela e no resultado de uma consulta. A SQL não elimina automaticamente tuplas repetidas nos resultados das consultas pelas seguintes razões:

- A eliminação de repetições é uma operação dispendiosa. Uma forma de implementá-la seria pela ordenação das tuplas primeiramente e, depois, pela eliminação das repetições.
- O usuário pode desejar ver as tuplas repetidas no resultado da consulta.
- Quando uma função de agregação (Seção 8.5.7) é aplicada nas tuplas, na maioria das vezes não se deseja eliminar as repetições.

8.4 Consultas SQL Básicas

161

Uma tabela SQL, com uma chave, restringe-se a um conjunto, uma vez que o valor da chave precisa ser diferente para cada tupla. Se, *de fato*, desejarmos eliminar as repetições no resultado de uma consulta SQL, precisaremos usar a palavra-chave **DISTINCT** na cláusula **SELECT**, o que fará com que somente as tuplas diferentes permaneçam no resultado. Em geral, uma consulta com **SELECT DISTINCT** elimina as repetições, enquanto uma consulta com **SELECT ALL** não o faz. Especificar **SELECT** sem **ALL** ou **DISTINCT** — como em nossos exemplos anteriores — é equivalente à opção **SELECT ALL**. Por exemplo, a Consulta 11 recupera o salário de todos os empregados; se alguns dos empregados tiverem o mesmo salário, o valor desse salário vai aparecer diversas vezes no resultado da consulta, como mostrado na Figura 8.4a. Se estivermos interessados somente nos diferentes valores dos salários, gostaríamos que cada valor aparecesse apenas uma vez, independentemente de quantos empregados ganhem esse salário. Pela palavra-chave **DISTINCT**, como na Q1 IA, chegamos a isto, como mostra a Figura 8.4b.

Consulta 11

Recupere o salário de todos os empregados (Q1 1) e todos os diferentes valores dos salários (Q11A).

```
Q11:  SELECT  ALL  SALÁRIO
FROM    EMPREGADO;
Q1 IA: SELECT  DISTINCT  SALÁRIO
FROM    EMPREGADO;
```

A SQL incorporou diretamente algumas operações de conjuntos da álgebra relacional: as operações de união de conjuntos (**UNION**), de diferença de conjuntos (**EXCEPT**) e de interseção de conjuntos (**INTERSECT**). As relações resultantes dessas operações de conjuntos são os conjuntos de tuplas, isto é, *as tuplas repetidas são eliminadas do resultado*. Como esse conjunto de operações é aplicado apenas em *relações compatíveis-por-união*, precisamos ter certeza de que as duas relações, nas quais vamos aplicar a operação, tenham os mesmos atributos, e que esses atributos apareçam na mesma ordem em ambas as relações. O próximo exemplo ilustra o uso da **UNION**.

Consulta 4

Faça uma lista com todos os números de projetos nos quais esteja envolvido algum empregado cujo último nome seja 'Smith'; ou como empregado, ou como gerente do departamento que controle o projeto.

```
Q4: (SELECT DISTINCT PNUMERO
FROM  PROJETO, DEPARTAMENTO, EMPREGADO
WHERE DNUM=DNUMERO AND GERSSN=SSN AND UNOME='Smith')
UNION
(SELECT DISTINCT PNUMERO
FROM  PROJETO, TRABALHA_EM, EMPREGADO
WHERE PNUMERO=PNO AND ESSN=SSN AND UNOME='Smith');
```

(*) SALÁRIO	(*) SALÁRIO
30000	30000
40000	40000
25000	25000
43000	43000
38000	38000
25000	55000 25000 55000
(c) PNUMO UNOME	(d) PNUMO UNOME
James Borg	

FIGURA 8.4 Resultado das consultas adicionais SQL, quando aplicadas ao banco de dados EMPRESA, no estado, mostrado na Figura 5.6. (a) Q11. (b) Q1 1 A. (c) Q16. (d) Q18.

A primeira consulta **SELECT** recupera os projetos que envolvem um 'Smith' como gerente do departamento que controla o projeto, enquanto a segunda recupera os projetos que envolvem um 'Smith' como um empregado que nele trabalhe. Note

8 Em geral, uma tabela SQL não impõe a existência de uma chave, embora, na maioria das vezes, ela exista.

162

Capítulo 8 SQL-99: Definição de Esquema, Restrições Básicas e Consultas (*Queries*)

que, se diversos empregados tiverem o último nome 'Smith', o nome do projeto que envolve algum deles será recuperado. A aplicação da operação UNION nas duas consultas SELECT fornece o resultado desejado.

A SQL também possui operações correspondentes às operações multiconjuntos, que serão seguidas pela palavra-chave ALL (UNION ALL, EXCEPT ALL, INTERSECT ALL). Seus resultados serão multiconjuntos (repetições não são eliminadas). Na Figura 8.5, o comportamento dessas operações é ilustrado com exemplos. Basicamente, quando se aplicam essas operações, cada tupla — sendo ela repetida ou não — é considerada uma tupla diferente.

(a)

R	A
	a1
	a2
	a2
	a3

(b)

T	A
	a1
	a1
	a2
	a2
	a2
	a3
	a4
	a5

(c)

S	A
	a1
	a2
	.a4
	a5

(d)

T	A
	a2
	a3

FIGURA 8.5 Os resultados das operações SQL multiconjunto. (a) Duas tabelas, R(A) e S(A). (b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

8.4.5 Comparações entre *Substrings* e Operadores Aritméticos

Nesta seção, discutiremos algumas outras funcionalidades da SQL. A primeira delas cria condições para comparações de partes de uma cadeia de caracteres por meio do operador de comparação LIKE. Esse operador pode ser usado para comparações de padrões de cadeias. As partes de cadeias podem ser especificadas usando-se dois caracteres reservados: % substitui um número arbitrário entre zero ou mais caracteres, enquanto *underscore* (_) substitui um único caractere. Por exemplo, considere a seguinte consulta:

Consulta 12

Recupere todos os empregados cujos endereços sejam em Houston, Texas.

Q12: **SELECT** PNOOME, LNOOME **FROM** EMPREGADO **WHERE** ENDEREÇO **LIKE** '%Houston,TX%';

Para recuperar todos os empregados que nasceram na década de 50, podemos usar a Consulta 12A. Aqui, '5' precisa ser o terceiro caractere da cadeia (de acordo com nosso formato de data), assim, usaremos o valor ' __5_____', com cada *underscore* servindo de substituto para um caractere qualquer.

Consulta 12A

Encontre todos os empregados que nasceram durante a década de 50.

Q12A: **SELECT FROM WHERE**

PNOME, UNOME EMPREGADO DATANASC LIKE

Se um sinal *underscore* ou % for necessário como um caractere literal em uma cadeia, ele deve ser precedido por um *caractere de escape*, o qual é especificado depois do *string* usando-se a palavra-chave ESCAPE. Por exemplo, 'AB_CD\%EF' ESCAPE 'V' representa a cadeia literal 'AB_CD%EF', porque \ é especificado como um caractere de escape. Qualquer outro

8.4 Consultas SQL Básicas

163

caractere que não seja usado na cadeia pode ser escolhido para o caractere de escape. Também precisamos de uma regra para especificar aspas simples ou duplas ("). Se elas tiverem de ser incluídas em uma cadeia de caracteres, uma vez que são usadas no começo e no fim das cadeias. Se forem necessárias aspas simples ('), elas devem ser representadas por aspas duplas ("), assim não serão interpretadas como final de cadeia.

Outra possibilidade é o uso de funções aritméticas em consultas. Os operadores aritméticos-padrão adição (+), subtração (-), multiplicação (*) e divisão (/) podem ser aplicados com valores numéricos ou atributos com domínios numéricos. Por exemplo, suponha que desejamos ver qual será o resultado de aumentar em 10% o salário de todos os empregados que trabalham com o projeto 'ProdutoX'; poderemos emitir a Consulta 13 para observar o que aconteceria com seus salários. Esse exemplo mostra também como podemos trocar o nome de um atributo, no resultado da consulta, usando AS na cláusula SELECT.

Consulta 13

Mostre o resultado dos salários caso fosse dado a todos os empregados que trabalham no 'ProductX' 10% de aumento.

Q13: **SELECT** PNAME, UNOME, 1.1 "SALÁRIO AS AUMENTO_SAL **FROM** EMPREGADO,TRABALHA_EM, PROJETO **WHERE** SSN=ESSN **AND** PNO=PNUMERO **AND** PNAME='ProdutoX';

Para as cadeias tipo data, o operador de concatenação || pode ser usado em uma consulta para juntar duas cadeias. Para os tipos data, horário, *timestamp* e intervalos de datas existem os operadores de incremento (+) e de decremento (-) de datas, de horário ou de *timestamp* para um intervalo. Além disso, um intervalo é o resultado da diferença entre duas datas, dois horários ou dois *timestamp*. Outro operador de comparação que pode ser usado por conveniência é BETWEEN (entre), que é ilustrado na Consulta 14.

Consulta 14

Recupere todos os empregados do departamento 5 que ganham entre 30 mil e 40 mil dólares.

Q14: **SELECT ***
FROM EMPREGADO
WHERE (SALÁRIO BETWEEN 30000 AND 40000) **AND** DNO = 5;
A condição (SALÁRIO BETWEEN 30000 AND 40000) em Q14 é equivalente à condição ((SALÁRIO >= 30000) **AND** (SALÁRIO <= 40000)).

8.4.6 Ordenando o Resultado das Consultas

A SQL permite que o usuário ordene as tuplas do resultado de uma consulta, pelos valores de um ou mais atributos, usando-se a cláusula ORDER BY. Isso é ilustrado na Consulta 15.

Consulta 15

Recupere a lista dos empregados e os respectivos projetos nos quais eles trabalham, ordenada por departamento, e, dentro de cada departamento, também por ordem alfabética do último nome, e depois, pelo primeiro nome do empregado.

Q15: **SELECT** DNAME, UNOME, PNAME, PJNAME
FROM DEPARTAMENTO, EMPREGADO, TRABALHAREM, PROJETO **WHERE** DNUMERO=DNO
AND SSN=ESSN **AND** PNO=PNUMERO **ORDER BY** DNAME, UNOME, PNAME;

A ordenação *default* é ascendente. Podemos especificar a palavra-chave DESC se quisermos que os valores do resultado apareçam na ordem descendente. A palavra-chave ASC pode ser usada para explicitar a ordenação ascendente. Por exemplo, se quisermos DNAME em ordem descendente e UNOME e PNAME em ordem ascendente, a cláusula ORDER BY de Q15 pode ser escrita como:

ORDER BY DNAME **DESC**, UNOME **ASC**, PNAME **ASC**

8.5 CONSULTAS SQL MAIS COMPLEXAS

Na seção anterior, descrevemos alguns dos tipos básicos de consultas em SQL. Em virtude do poder de generalização e da expressividade da linguagem, existem muitas funcionalidades adicionais que permitem ao usuário definir as consultas mais complexas. Discutiremos algumas delas nesta seção.

8.5.1 Comparações Envolvendo NULL e os Três Valores Lógicos

A SQL possui várias regras para o tratamento dos valores NULL. Relembrando a Seção 5.1.2, o NULL é usado para representar os valores inexistentes, mas normalmente tem uma das três interpretações — valor desconhecido (existe, mas não é conhecido), valor indisponível (existe, mas é propositadamente omitido) ou não aplicável (indefinido para a tupla em questão). Considere os seguintes exemplos para ilustrar cada um dos três significados de NULL:

1. *Valor desconhecido*: não se conhece a data do aniversário de uma pessoa em particular, assim, essa data será representada por NULL no banco de dados.
2. *Valores indisponíveis ou omitidos*: uma pessoa possui um telefone, mas não quer que seja armazenado; dessa forma, o número será omitido e representado por NULL no banco de dados.
3. *Não aplicável ao atributo*: um atributo UltimoGrauEscolar pode ser NULL para uma pessoa que não possui nenhum grau escolar, logo, ele não se aplica àquela pessoa.

Freqüentemente, não é possível determinar qual dos três sentidos é o pretendido; por exemplo, um NULL para o número do telefone da casa de uma pessoa pode possuir qualquer um dos três significados. Logo, a SQL não faz distinção entre os diferentes significados de NULL.

Em geral, cada NULL é considerado diferente de outro NULL dentro do banco de dados. Quando um NULL é comparado em uma operação, o resultado é considerado UNKNOWN [desconhecido — pode ser tanto TRUE (verdadeiro) quanto FALSE (falso)]. Assim, a SQL usa três valores lógicos TRUE (verdadeiro), FALSE (falso) e UNKNOWN (desconhecido) em vez dos dois valores lógicos-padrão, TRUE e FALSE. É necessário, portanto, definir as expressões para os resultados com os três valores lógicos quando se usam conectivos lógicos como AND, OR e NOT. A Tabela 8.1 mostra os valores resultantes.

TABELA 8.1 Conectivos Lógicos com os Três Valores Lógicos

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN
OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
NOT			
TRUE	FALSE		
FALSE	TRUE		
UNKNOWN	UNKNOWN		

Em consultas *select-project-join*, a regra é que somente as combinações de tuplas que evoluírem para uma expressão lógica TRUE (verdadeira) da consulta serão selecionadas. As combinações de tuplas que evoluírem para FALSE (falso) ou UNKNOWN (desconhecido) não serão selecionadas. Entretanto, existem exceções para essa regra em certas operações, como junções externas (*outer join*), como vamos ver.

8.5 Consultas SQL Mais Complexas

165

A SQL permite consultas que chequem se o valor de um atributo é NULL. Para comparar o valor de um atributo com NULL, em vez de usar = ou <>, a SQL usa IS (é) ou IS NOT (não é). Isso porque a SQL considera cada valor NULL diferente de todos os outros valores NULL; logo, a comparação pela igualdade não é apropriada. Quando uma condição para junção é especificada, as tuplas com os valores NULL nos atributos de junção não são incluídas no resultado (salvo se a junção for externa, OUTER JOIN; Seção 8.5.6). Isso é ilustrado na Consulta 18, cujo resultado é apresentado na Figura 8.4d.

Consulta 18

Recupere os nomes de todos os empregados que não têm supervisor.

Q18: **SELECT** PNAME, UNOME **FROM** EMPREGADO **WHERE** SUPERSSN IS NULL;

8.5.2 Consultas Aninhadas, Comparações de Tuplas e de Conjuntos/Multiconjuntos

Algumas consultas necessitam da busca de valores presentes no banco de dados para, então, usá-los na condição de comparação. Essas consultas podem ser formuladas de modo conveniente por meio de **consultas aninhadas** (*nested queries*), que formam um bloco completo de *select-from-where* dentro da cláusula WHERE de outra consulta. Essa outra consulta é chamada **consulta externa** (*outer query*). A Consulta 4 foi formulada sem o uso de consultas aninhadas, mas poderia ser refeita utilizando-as como mostra a Q4A. A Q4A introduz o operador de comparação IN, que compara um valor *v* com um conjunto (ou multiconjuntos) de valores *V* e evolui para verdadeiro, TRUE, se *v* for um dos elementos de *V*.

Q4A: **SELECT** **DISTINCT** PNUMERO **FROM** PROJETO **WHERE** PNUMERO **IN** (**SELECT** PNUMERO

FROM PROJETO, DEPARTAMENTO, EMPREGADO **WHERE** DNUM=DNUMEROAND
GERSSN=SSN AND UNOME='Smith') **OR** PNUMERO **IN** (**SELECT** PNO

FROM TRABALHA_EM, EMPREGADO **WHERE** ESSN=SSN AND UNOME='Smith');

A primeira consulta aninhada seleciona os números de projetos que tiverem um 'Smith' como supervisor, enquanto a segunda seleciona os números de projetos que tiverem um empregado 'Smith' envolvido. Na consulta externa, usamos o conectivo lógico OR (ou) para recuperar uma tupla de PROJETO cujo valor PNUMERO esteja contido no resultado de uma das consultas aninhadas. Se a consulta aninhada devolve um atributo único e uma tupla única, o resultado da consulta será um valor único (escalar). Nesses casos, é possível usar = em vez de IN como operador de comparação. De modo geral, a consulta aninhada vai devolver uma **tabela** (relação), que é um conjunto ou um multiconjunto de tuplas.

A SQL permite o uso de **tuplas** de valores em comparações colocando-as entre parênteses. Para ilustração, considere a seguinte consulta:

SELECT **DISTINCT** ESSN **FROM** TRABALHA_EM
WHERE (PNO, HORAS) **IN** (**SELECT** PNO, HORAS **FROM** TRABALHA_EM
WHERE SSN='123456789');

Essa consulta vai selecionar o número do seguro social de todos os empregados que trabalham com a mesma combinação (projeto, horas) em algum dos projetos em que o empregado 'John Smith' (SSN= '123456789') trabalhe. Nesse exemplo, o operador IN compara a sub tupla de valores entre parênteses (PNO, HORAS) de cada tupla de TRABALHA_EM com p conjunto de tuplas compatíveis-por-união produzido pela consulta aninhada.

Capítulo 8 SQL-99: Definição de Esquema, Restrições Básicas e Consultas (*Queries*)

Além do operador IN, outros operadores de comparação podem ser usados para comparar um único valor v (normalmente um nome de atributo) a um conjunto ou multiconjunto V (geralmente uma consulta aninhada). O operador $=$ ANY (ou $=$ SOME) devolve TRUE se o valor v for igual a *algum valor* do conjunto V , assim é equivalente ao operador IN. As palavras-chave ANY e SOME possuem o mesmo significado (em português, 'algum'). Outros operadores podem ser combinados com ANY (ou SOME), incluindo $>$, $>=$, $<$, $<=$ e $<>$. A palavra-chave ALL pode ser combinada com cada um desses operadores. Por exemplo, a condição de comparação $(v > ALL V)$ devolve TRUE se o valor v for maior que *todos* os valores do conjunto (ou multiconjunto) V . Um exemplo é a consulta seguinte que devolve os nomes dos empregados cujos salários são maiores que os salários de todos os empregados do departamento 5:

```
SELECT UNOME, PNOME FROM EMPREGADO
WHERE SALÁRIO > ALL (SELECT SALÁRIO FROM EMPREGADO
WHERE DNO=5);
```

Em geral, podemos definir vários níveis de consultas aninhadas. Novamente, podemos nos defrontar com possíveis ambigüidades entre os nomes de atributos, se existirem atributos de mesmo nome — um na relação especificada na cláusula FROM da *consulta externa*, e outro na relação especificada na cláusula FROM da *consulta aninhada*. A regra é que a referência a um atributo *não qualificado* é atribuída à relação declarada na **consulta mais interna** do ninho de consultas. Por exemplo, na cláusula SELECT e WHERE da primeira consulta aninhada de Q4A, a referência a qualquer atributo não qualificado da relação PROJETO é atribuída à relação PROJETO especificada na cláusula FROM da consulta aninhada. Para fazer referência a um atributo da relação PROJETO, estabelecida na consulta externa, precisaremos especificar, para ser usado como referência, um *alias* (variável de tupla) daquela relação. Essas regras são similares às utilizadas na maioria das linguagens de programação, em variáveis de programas, que permitem a utilização de procedimentos e funções aninhadas. Para ilustrar a ambigüidade potencial dos nomes de atributos em consultas aninhadas, considere a Consulta 16, cujo resultado é apresentado na Figura 8.4c.

Consulta 16

Recupere o nome de cada um dos empregados que tenham um dependente cujo primeiro nome e sexo sejam o mesmo do empregado em questão.

```
Q16: SELECT E.PNOME, E.UNOME FROM EMPREGADO AS E WHERE E.SSNIN (SELECT
ESSN
FROM DEPENDENTE WHERE E.PNOME=DEPENDENTE_NOME AND E.SEXO=SEXO);
```

Na consulta aninhada Q16, precisamos qualificar E.SEXO porque ele se refere ao atributo SEXO do EMPREGADO da consulta externa, e DEPENDENTE tem também um atributo chamado SEXO. Todas as referências não qualificadas de SEXO na consulta aninhada referem-se ao SEXO do DEPENDENTE. Entretanto, *não precisamos* qualificar PNOME e SSN porque a relação DEPENDENTE não possui atributos chamados PNOME e SSN; logo, não há nenhuma ambigüidade.

Normalmente é recomendável a criação de variáveis de tuplas (*aliases*) para *todas as tabelas referidas por uma consulta SQL*, de modo a evitar erros e ambigüidades em potencial.

8.5.3 Consultas Aninhadas Correlacionadas

Sempre que uma condição na cláusula WHERE de uma consulta aninhada se referir a algum atributo da relação declarada na consulta externa, as duas consultas são chamadas **correlacionadas**. Podemos compreender melhor a correlação entre consultas se considerarmos que *a consulta aninhada é avaliada uma vez para cada tupla (ou para uma combinação de tuplas) da consulta externa*. Por exemplo, podemos imaginar Q16 como segue: para *cada* tupla de EMPREGADO, a consulta aninhada avalia para recuperar os valores dos ESSN de todas as tuplas de DEPENDENTE que possuem o mesmo sexo e o mesmo nome da tupla EMPREGADO em questão; se o valor do SSN da tupla EMPREGADO está *no* resultado da consulta aninhada, então essa tupla EMPREGADO será selecionada.

Em geral, uma consulta escrita com blocos *select/from/where* aninhados e usando os operadores de comparação $=$ ou IN poderá *sempre* ser expressa como um bloco único de consulta. Por exemplo, a Q16 pode ser escrita como a Q16A a seguir:

8.5 Consultas SQL Mais Complexas 167

Q16A: **SELECT** E.PNOME, E.UNOME

FROM EMPREGADO **AS** E, DEPENDENTE **AS** D **WHERE** E.SSN=D.ESSN **AND** E.SEXO=D.SEXO **AND** E.PNOME=D.DEPENDENTE_NOME;

A implementação original da SQL no SISTEMA R tinha, também, o operador de comparação CONTAINS, que era usado para comparar dois conjuntos ou multiconjuntos. Esse operador foi eliminado da linguagem, possivelmente em razão da dificuldade de implementação eficiente. Implementações mais comerciais da SQL *não* têm esse operador. O operador CONTAINS compara dois conjuntos de valores e devolve TRUE se um conjunto contiver todos os valores do outro conjunto. A Consulta 3 ilustra o uso do operador CONTAINS.

Consulta 3

Recupere o nome de cada um dos empregados que trabalha em todos os projetos controlados pelo departamento 5.

Q3: **SELECT** PNOME, UNOME **FROM** EMPREGADO

WHERE (**SELECT** PNO

FROM TRABALHA_EM

WHERE SSN=ESSN)

CONTAINS

(**SELECT** PNUMERO

FROM PROJETO

WHERE DNUM=5));

Em Q3, a segunda consulta aninhada (que não está correlacionada à consulta externa) devolve os números de todos os projetos controlados pelo departamento 5. Para *cada* tupla de empregado, a primeira consulta aninhada (que está correlacionada) devolve o número do projeto nos quais os empregados trabalham; se eles contiverem todos os projetos controlados pelo departamento 5, a tupla empregado será selecionada e o nome daquele empregado, recuperado. Observe que o operador de comparação CONTAINS possui função similar à operação DIVISION da álgebra relacional (Seção 6.3.4) e à quantificação universal do cálculo relacional (Seção 6.6.6). Como a operação CONTAINS não é parte da SQL, temos de usar outras técnicas, como a função EXISTS, para especificar esses tipos de consultas, conforme descrito na Seção 8.5.4.

8.5.4 As Funções EXISTS e UNIQUE da SQL

A função EXISTS da SQL é usada para verificar se o resultado de uma consulta aninhada correlacionada é vazio (não contém nenhuma tupla) ou não. Ilustramos o uso do EXISTS — e do NOT EXISTS — com alguns exemplos. Primeiro, formulamos um modo alternativo para a Consulta 16 usando EXISTS. A Q16B mostra essa opção:

Q16B: **SELECT** E.PNOME, E.UNOME **FROM** EMPREGADO **AS** E **WHERE** EXISTS (**SELECT***

FROM DEPENDENTE

WHERE E.SSN=ESSN **AND** E.SEXO=SEXO

AND E.PNOME=DEPENDENTE_NOME);

EXISTS e NOT EXISTS são empregados, normalmente, em conjunto com as consultas aninhadas correlacionadas. Na Q16B, a consulta aninhada faz referência aos atributos SSN, PNOME e SEXO da relação EMPREGADO da consulta externa. Podemos imaginar a Q16B como segue: para cada tupla EMPREGADO, a consulta aninhada avalia devolvendo todas as tuplas de DEPENDENTE com o mesmo número de seguro social, sexo e nome da tupla EMPREGADO; se existir (EXISTS) pelo menos uma tupla no resultado da consulta aninhada, então a tupla EMPREGADO em questão será selecionada. Em geral, EXISTS (Q) devolverá TRUE se existir *ao menos uma tupla* no resultado da consulta aninhada Q, caso contrário, devolverá FALSE. Porém, NOT EXISTS (Q) devolverá TRUE se não existir *nenhuma tupla* no resultado da consulta aninhada Q, caso contrário, devolverá FALSE. A seguir, ilustramos o uso

de

NOT

EXISTS.

168

Capítulo 8 SQL-99: Definição de Esquema, Restrições Básicas e Consultas (*Queries*)**Consulta 6**

Recupere os nomes dos empregados que não possuem nenhum dependente.

Q6: SELECT FROM WHERE

PNOME, UNOME

EMPREGADO

NOT EXISTS (SELECT FROM WHERE

DEPENDENTE SSN=ESSN);

Em Q6, a consulta aninhada correlacionada devolve todas as tuplas de DEPENDENTE relacionadas a uma tupla EMPREGADO em particular. Se não existir *nenhuma*, a tupla EMPREGADO será selecionada. Podemos explicar Q6 como segue: para *cada* tupla EMPREGADO, a consulta aninhada correlacionada seleciona todas as tuplas de DEPENDENTE cujos valores ESSN coincidirem com o valor do SSN do EMPREGADO; se o resultado for vazio, nenhum dependente está relacionado ao empregado, então, selecionamos a tupla EMPREGADO e recuperamos seu NOME e UNOME.

Consulta 7

Relacione os nomes dos gerentes que possuam ao menos um dependente.

Q7: SELECT FROM WHERE

PNOME, UNOME

EMPREGADO

**EXISTS (SELECT
FROM WHERE AND EXISTS
DEPENDENTE SSN=ESSN)**
(SELECT**FROM****WHERE**

DEPARTAMENTO SSN=GERSSN);

Um jeito de escrever essa consulta é mostrado em Q7, na qual especificamos duas consultas aninhadas e correlacionadas; a primeira seleciona todas as tuplas de DEPENDENTE relacionadas a EMPREGADO e a segunda seleciona todas as tuplas de DEPARTAMENTO gerenciadas por um EMPREGADO. Se ao menos uma da primeira e uma da segunda existirem, selecionamos a tupla EMPREGADO. Você poderia reformular essa consulta usando apenas uma única consulta aninhada ou nenhuma consulta aninhada?

A Consulta 3 ('Recupere o nome de cada um dos empregados que trabalha em *todos* os projetos controlados pelo departamento 5', Seção 8.5.3) pode ser estabelecida usando EXISTS e NOT EXISTS em sistemas SQL. Existem duas opções. A primeira é usar a transformação da muito bem conhecida transformação da teoria dos conjuntos, em que (*S1 CONTAINS S2*) é logicamente equivalente a (*S2 EXCEPTS S1*) se ele for vazio. A Q3A mostra essa opção.

Q3A:**SELECT** PNOME, UNOME**FROM** EMPREGADO**WHERE NOT EXISTS**
**((SELECT PNUMERO
FROM PROJETO
WHERE DNUM=5)**
EXCEPT**(SELECT PNO****FROM TRABALHA_EM****WHERE SSN=ESSN));**

Em Q3A, a primeira subconsulta (que não está correlacionada) seleciona todos os projetos controlados pelo departamento 5, e a segunda (que está correlacionada) seleciona todos os projetos nos quais o empregado em particular, que está sendo considerado, trabalhe. Se a diferença dos conjuntos, o da primeira subconsulta MINUS (EXCEPT) e o da segunda, for vazio, isso significa que o empregado trabalha em todos os projetos e então é selecionado.

9 Lembre-se de que EXCEPT é o operador da diferença entre os conjuntos.

8.5 Consultas SQL Mais Complexas

169

A segunda opção é mostrada em Q3B. Observe que precisamos de dois níveis de aninhamento em Q3B e que essa formulação é um pouco mais complexa que na Q3, que usa o operador de comparação CONTAINS, e que na Q3 A, que utiliza NOT EXISTS e EXCEPT. Entretanto, CONTAINS não faz parte da SQL nem todos os sistemas relacionais possuem o operador EXCEPT, embora ele faça parte da SQL-99.

Q3B:

```

SELECT    UNOME, PNAME
FROM      EMPREGADO
WHERE      NOT EXISTS
(
SELECT      *
FROM        TRABALHA_EM E
WHERE      (B.PNO IN      (SELECT      PNUMERO
                                FROM        PROJETO
                                WHERE      DNUM=5) )

          AND
          NOT EXISTS      (SELECT      *
                                FROM        TRABALHA_EM C
                                WHERE      C.ESSN=SSN
                                AND        C.PNO=B.PNO) );

```

Em Q3B, a consulta aninhada mais externa seleciona qualquer tupla TRABALHA_EM (B) cujos PNO sejam de projetos controlados pelo departamento 5, se não existir nenhuma tupla TRABALHA_EM (C) com o mesmo PNO e o mesmo SSN da tupla EMPREGADO que está sendo considerada na consulta externa. Se não existir nenhuma dessas tuplas, selecionaremos a tupla EMPREGADO. A forma usada em Q3B combina com a seguinte reformulação da Consulta 3: Selecione cada empregado de maneira que não exista um projeto controlado pelo departamento 5 que o empregado não trabalhe. Isso corresponde ao modo pelo qual formulamos essa consulta em cálculo relacional de tuplas, na Seção 6.6.6.

Existe outra função SQL, UNIQUE(Q), que devolve TRUE se não existir nenhuma tupla repetida no resultado da consulta Q; caso contrário, ela devolverá FALSE. Isso pode ser usado para verificar se o resultado da consulta aninhada é um conjunto ou um multiconjunto.

8.5.5 Conjuntos Explícitos e Nomes Alternativos de Atributos em SQL

Vimos diversas consultas aninhadas na cláusula WHERE. E também possível usar um **conjunto de valores explícitos** na cláusula WHERE em vez de consultas aninhadas. Esse conjunto, em SQL, é colocado entre parênteses.

Consulta 17

Recupere os números dos seguros sociais de todos os empregados que trabalham nos projetos 1, 2 ou 3.

DISTINCT ESSN TRABALHA_EM PNO IN (1,2, 3);

Q17: **SELECT FROM WHERE**

Em SQL é possível dar nomes alternativos a qualquer atributo pela adição do qualificador AS seguido do novo nome desejado — esses nomes aparecerão no resultado da consulta. Assim, o construtor AS pode ser usado como *alias* (pseudônimo) tanto nos nomes dos atributos quanto nos das relações; pode ser utilizado também em ambas as cláusulas SELECT e FROM. Por exemplo, Q8A mostra como a Consulta 8 pode ser ligeiramente alterada para recuperar o último nome de cada empregado e seu(sua) supervisor(a), e denominar os atributos no resultado como NOME_EMPREGADO e NOME_SUPERVISOR. Os novos nomes vão aparecer como títulos nas colunas do resultado da consulta.

Q8A: **SELECT** E.UNOME AS NOME_EMPREGADO, S.UNOME AS NOME_SUPERVISOR **FROM** EMPREGADO AS E, EMPREGADO AS S **WHERE** E.SUPERSSN=S.SSN;

8.5.6 Junção (*Joiri*) de Tabelas em SQL

O conceito de **junção de tabelas** (ou **junção de relações**) foi incorporado à SQL para que os usuários pudessem especificar uma tabela que fosse resultado da aplicação da operação de junção na cláusula FROM de uma consulta. Esse construtor pode ser mais bem compreendido quando se misturam todas as condições de seleção e de junção na cláusula WHERE. Por exemplo, considere a consulta Q1, que recupera o nome e o endereço de todos os empregados que trabalham no departamento 'Pesquisa'. Pode ser mais fácil especificar primeiro a junção das relações EMPREGADO e DEPARTAMENTO, e depois, selecionar as tuplas e os atributos. Essa forma pode ser escrita em SQL como a Q1A:

Q1A: **SELECT** P.NOME, U.NOME, ENDEREÇO

FROM (EMPREGADO **JOIN** DEPARTAMENTO **ON** DNO=DNUMERO) **WHERE** D.NOME='Pesquisa';

A cláusula FROM em Q1 A contém uma única *tabela juntada* (*joined table*). Os atributos dessa tabela são todos aqueles da primeira tabela, EMPREGADO, seguidos de todos os atributos da segunda tabela, DEPARTAMENTO. O conceito de junção de tabela também permite que o usuário especifique os tipos de junções diferentes, como NATURAL JOIN (junção natural) e vários tipos de OUTER JOIN (junção externa). Na NATURAL JOIN de duas relações R e S, nenhuma condição de junção é especificada; uma condição equijunção (*equijoin*) implícita é criada para cada par de atributos com o mesmo nome. Cada um desses pares de atributos é incluído somente uma vez na relação resultante (Seção 6.4.3).

No caso de os nomes dos atributos de junção não serem os mesmos das relações básicas, é possível atribuir novos nomes aos atributos de forma que combinem, e então aplicar a NATURAL JOIN. Nesse caso, o construtor AS pode ser usado para mudar o nome da relação e de todos os seus atributos na cláusula FROM. A consulta Q1B ilustra essa condição em que a relação DEPARTAMENTO recebe o nome de DEPT e seus atributos recebem os nomes de D.NOME, DNO (para combinar com o nome do atributo DNO de EMPREGADO escolhido como atributo de junção), GERSSN e GERDATAIN. A condição para a junção, usada nesse caso de NATURAL JOIN, foi EMPREGADO.DNO = DEPT.DNO, porque esse é o único par de atributos com o mesmo nome depois da nova designação.

Q1B: **SELECT** P.NOME, U.NOME, ENDEREÇO **FROM** (EMPREGADO **NATURAL JOIN**

(DEPARTAMENTO **AS** DEPT (D.NOME, DNO, GERSSN, GERDATAIN))) **WHERE** D.NOME='Pesquisa';

O tipo *default* de junção de uma tabela é a **junção interna** (*innerjoin*), na qual uma tupla é incluída no resultado somente se existir uma tupla que combine na outra relação. Por exemplo, na consulta Q8A, somente os empregados que *possuírem um supervisor* serão incluídos no resultado; uma tupla EMPREGADO, cujo valor de SUPERSSN for NULL, será excluída. Se o usuário quiser que todos os empregados sejam incluídos, é necessário explicitar um OUTER JOIN (Seção 6.4-3 para a definição de OUTER JOIN). Em SQL isso pode ser tratado por meio da especificação explícita do OUTER JOIN em uma junção de tabela, como ilustrado em Q8B:

Q8B: **SELECT** E.UNOME **AS** NOME_EMPREGADO, S.UNOME **AS** NOME_SUPERVISOR **FROM** (EMPREGADO **AS** E **LEFT OUTER JOIN** EMPREGADO **AS** S **ON** E.SUPERSSN=S.SSN);

As opções para a especificação de junção de tabelas disponíveis em SQL incluem INNER JOIN (mesmo que JOIN), LEFT OUTER JOIN (junção externa à esquerda), RIGHT OUTER JOIN (junção externa à direita) e FULL OUTER JOIN (junção externa completa). Nas três últimas opções, a palavra-chave OUTER pode ser omitida. Se os atributos de junção possuírem o mesmo nome, um deles também pode especificar a variação junção natural das junções externas por intermédio do uso da palavra-chave NATURAL antes da operação (por exemplo, NATURAL LEFT OUTER JOIN). A palavra-chave CROSS JOIN é usada para especificar a operação produto cartesiano (Seção 6.2.2), embora possa ser usada somente com o máximo de cuidado possível, porque são gerados todos os tipos de combinações de tuplas possíveis.

Também é possível especificar junções *aninhadas* (*nestjoin*), isto é, uma das tabelas em uma junção é, ela própria, fruto de junção de tabelas. Isso é ilustrado em Q2A, que é uma alternativa para a construção da consulta Q2 usando o conceito de junção de tabela:

Q2A: **SELECT** P.NUMERO, DNUM, U.NOME, ENDEREÇO, DATANASC

FROM ((PROJETO **JOIN** DEPARTAMENTO **ON** DNUM=DNUMERO)

8.5 Consultas SQL Mais Complexas

171

JOIN EMPREGADO ON GERSSN=SSN) WHERE PLOCALIZACAO='Stafford';

8.5.7 Funções Agregadas em SQL

Na Seção 6.4-1, introduzimos o conceito de função agregada como uma operação relacional. Como o agrupamento e a agregação são necessários em diversas aplicações de um banco de dados, a SQL tem funcionalidades que incorporam esses conceitos. Há diversas funções pré-construídas para esse fim: COUNT, SUM, MAX, MIN e AVG. A função COUNT devolve o número de tuplas ou valores especificado em uma consulta. As funções SUM, MAX, MIN e AVG são aplicadas em um conjunto ou multi-conjunto de valores e devolvem, respectivamente, a soma, o valor máximo, o valor mínimo e a média desses valores. Essas funções podem ser usadas na cláusula SELECT ou na cláusula HAVING (que apresentaremos mais tarde). As funções MAX e MIN podem ser usadas também com atributos que não tenham domínios numéricos, se os domínios desses valores tiverem uma *ordenação total* entre eles. Ilustraremos o uso dessas funções com exemplos de consultas.

Consulta 19

Encontre a soma dos salários, o maior salário, o menor salário e a média salarial de todos os empregados.

Q19: **SELECT SUM (SALÁRIO), MAX (SALÁRIO), MIN (SALÁRIO), AVG (SALÁRIO) FROM EMPREGADO;**

Se quisermos obter os valores das funções anteriores para os empregados de um departamento específico — digamos, do departamento de 'Pesquisa' —, podemos formular a Consulta 20, na qual as tuplas de EMPREGADO são discriminadas pela cláusula WHERE para aqueles empregados que trabalham no departamento de 'Pesquisa'.

Consulta 20

Encontre a soma dos salários de todos os empregados do departamento 'Pesquisa', bem como o maior salário, o menor salário e a média salarial desse departamento.

Q20: **SELECT SUM (SALÁRIO), MAX (SALÁRIO), MIN (SALÁRIO), AVG (SALÁRIO) FROM (EMPREGADO JOIN DEPARTAMENTO ON DNO=DNUMERO) WHERE DNOME='Pesquisa';**

Consultas 21 e 22

Recupere o número total de empregados da empresa (Q21) e o número de empregados do departamento 'Pesquisa' (Q22).

Q21: **SELECT COUNT (*)**

FROM EMPREGADO ;

Q22: **SELECT COUNT (*)**

FROM EMPREGADO, DEPARTAMENTO WHERE DNO=DNUMERO AND DNOME= 'Pesquisa' ;

Aqui, o asterisco (*) refere-se às *linhas* (tuplas), logo, COUNT (*) devolverá, para o resultado da consulta, o número de linhas. Também podemos usar a função COUNT para contar o número de valores em uma coluna em vez do número de tuplas, como veremos no próximo exemplo.

10 Foram inseridas na SQL-99 as funções de agregação adicionais para os cálculos estatísticos mais avançados.

11 Ordenação total significa que, para quaisquer dois valores de um domínio, pode ser estabelecido que um apareça antes do outro em uma ordem definida; por exemplo, os domínios de DATE, TIME e TIMESTAMP têm ordenação total em seus valores, bem como as cadeias de caracteres alfabéticos.

Consulta 23

Conte o número dos diferentes valores de salário contidos no banco de dados.

Q23: SELECT COUNT (DISTINCT SALÁRIO) FROM EMPREGADO;

Se escrevermos COUNT(SALÁRIO) em vez de COUNT(DISTINCT SALÁRIO) na Q23, então os valores repetidos não serão eliminados. Entretanto, uma tupla com NULL em SALÁRIO não será contabilizada. Em geral, os valores NULL são **descartados** quando se aplicam as funções agregadas em uma coluna (atributo) em particular.

O exemplo anterior resume a *relação total* (Q19, Q21 e Q23) ou seleciona um subconjunto de tupla (Q20, Q22), portanto, produzem as tuplas únicas ou os valores únicos. Elas ilustram como a função é aplicada para recuperar os valores totalizados ou as tuplas totalizadas em um banco de dados. Essas funções também podem ser utilizadas em condições de seleção envolvendo as consultas aninhadas. Podemos especificar uma consulta aninhada correlacionada com funções agregadas e, então, usar a consulta aninhada dentro de uma cláusula WHERE da consulta externa. Por exemplo, para recuperar os nomes de todos os empregados que têm dois ou mais dependentes (Consulta 5), podemos fazer o seguinte:

Q5: SELECT UNOME, PNOME FROM EMPREGADO WHERE (SELECT COUNT (*) FROM DEPENDENTE WHERE SSN=ESSN) >= 2;

A tupla aninhada correlacionada conta o número de dependentes que cada um dos empregados tem; se esse número for maior ou igual a dois, a tupla desse empregado será selecionada.

8.5.8 Agrupamento: as Cláusulas GROUP BY e HAVING

Em muitos casos, precisamos aplicar as funções agregadas *para os subgrupos de tuplas de uma relação*, na qual os subgrupos são escolhidos com base em alguns atributos, por exemplo, quando precisamos encontrar a média de salário dos empregados *de cada departamento* ou o número de empregados que trabalham *em cada projeto*. Nesses casos, precisaremos **particionar** a relação em subconjuntos (ou **grupos**) de tuplas que não se sobreponham. Cada grupo (partição) consistirá em tuplas que tenham o mesmo valor em algum(ns) de seu(s) atributo(s), chamado(s) **atributo(s) de agrupamento**. Podemos aplicar a função para cada um desses grupos de atributos, independentemente. A SQL possui a cláusula GROUP BY para esse fim. Essa cláusula especifica os atributos de agrupamento, os quais poderiam *também aparecer na cláusula SELECT*, assim, o valor resultante da aplicação de cada função agregada no grupo de tuplas aparecerá com o valor do(s) atributo(s) de agrupamento.

Consulta 24

Para cada departamento, recupere seu número, o número de empregados que nele trabalham e a média de seus salários.

Q24: SELECT DNO, COUNT (*), AVG (SALÁRIO) FROM EMPREGADO GROUP BY DNO;

Em Q24, as tuplas EMPREGADO serão particionadas em grupos — cada grupo tendo o mesmo valor no atributo de agrupamento DNO. As funções COUNT e AVG serão aplicadas em cada grupo de tuplas. Note que a cláusula SELECT inclui somente o atributo de agrupamento e as funções que serão aplicadas em cada um dos grupos de atributos. A Figura 8.6a ilustra como o agrupamento age na Q24; também mostra o resultado de Q24.

Se existir NULL no atributo de agrupamento, então um **grupo separado** será criado para todas as tuplas com *valores NULL no atributo de agrupamento*. Por exemplo, se a tabela EMPREGADO contiver algum atributo que tenha NULL no atributo de agrupamento DNO, poderia haver um grupo separado para essas tuplas no resultado de Q24.

Consulta 25

Para cada projeto, recupere seu número, seu nome e o número de empregados que nele trabalham.

8.5 Consultas SQL Mais Complexas

173

(a)

PNOME	MINICIAL	LNOME	SSN		SALÁRIO	SUPERSSN	DNO
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453		25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbat	987987987		25000	987654321	4
James	E	Bong	888665555		55000	null	1

Agrupamento das tuplas EMPREGADO por meio do valor de DNO

(b)

PNOME	PNUMERO		ESSN	PNO	HORAS
ProdutoX	1		123456789	1	32,5
ProdutoX	1		453453453	1	20,0
ProdutoY	2		123456789	2	7,5
ProdutoY	2		453453453	2	20,0
ProdutoY	2		333445555	2	10,0
ProdutoZ	3		666884444	3	40,0
ProdutoZ	3		333445555	3	10,0
Automação	10		333445555	10	10,0
Automação	10		999887777	10	10,0
Automação	10		987987987	10	35,0
Reorganização	20		333445555	20	10,0
Reorganização	20		987654321	20	15,0
Reorganização	20		888665555	20	null
NovosBeneficios	30		987987987	30	5,0
NovosBeneficios	30		987654321	30	20,0
NovosBeneficios	30		999887777	30	30,0

	DNO	COUNT (*)	AVG (SALÁRIO)
5	4		33250
4	3		31000
1	1		55000

Resultado da Q24

Esses grupos não são selecionados por HAVING condição da 0.26

Depois da aplicação da cláusula WHERE, mas antes da aplicação da cláusula HAVING

PNOME	PNUMERO		ESSN	PNO	HORAS
ProdutoY	2		123456789	2	7,5
ProdutoY	2		453453453	2	20,0
ProdutoY	2		333445555	2	10,0
Automação	10		333445555	10	10,0
Automação	10		999887777	10	10,0
Automação	10		987987987	10	35,0
Reorganizacao	20		333445555	20	10,0
Reorganizacao	20		987654321	20	15,0
Reorganizacao	20		888665555	20	null
NovosBeneficios	30		987987987	30	5,0
NovosBeneficios	30		987654321	30	20,0
NovosBeneficios	30		999887777	30	30,0
PNOME	COUNT (*)				
ProdutoY	3				
Automação	3				
Reorganização	3				
NovosBeneficios	3				

Resultado da Q26 (PNUMERO não apresentado)

Depois da aplicação da condição da cláusula HAVING **FIGURA 8.6** Resultado da GROUP BY e da HAVING. (a) Q24. (b) Q26.

Q25: SELECT PNUMERO, PNOME, COUNT (*) FROM PROJETO, TRABALHA_EM WHERE PNUMERO=PNO GROUP BY PNUMERO, PNOME;

Q25 mostra como podemos usar uma condição de junção com a GROUP BY. Nesse caso, o agrupamento e as funções serão aplicados *depois* da junção das duas relações. Algumas vezes queremos recuperar os valores dessas funções somente para os *grupos*

que satisfazem certas condições. Por exemplo, suponha que queiramos modificar a Consulta 25 de modo que somente os projetos com mais de dois empregados apareçam no resultado. Para esse fim, a SQL oferece a cláusula HAVING, que pode aparecer em conjunto com a cláusula GROUP BY. HAVING proporciona a aplicação de uma condição para o grupo de tuplas associado a cada valor dos atributos de agrupamento. Apenas os grupos que satisfizerem a condição serão apresentados no resultado da consulta. Isso é ilustrado na Consulta 26.

Consulta 26

Para cada projeto *em que trabalhem mais de dois empregados*, recupere o número do projeto, seu nome e o número de empregados.

```
Q26: SELECT      PNUMERO, PJNOME, COUNT (*)
FROM            PROJETO, TRABALHA_EM
WHERE           PNUMERO=PNO
GROUP BY       PNUMERO, PJNOME
HAVING         COUNT (*) > 2;
```

Observe que, enquanto as condições de seleção da cláusula WHERE limitam as *tuplas* nas quais as funções serão aplicadas, a cláusula HAVING serve para escolher *grupos inteiros*. A Figura 8.6b ilustra o uso da HAVING e mostra o resultado da Q26.

Consulta 27

Para cada projeto, recupere seu número, seu nome e o número de empregados do departamento 5 que nele trabalhem.

```
Q27: SELECT      PNUMERO, PNAME, COUNT (*)
FROM            PROJETO, TRABALHA_EM, EMPREGADO WHERE PNUMERO=PNO AND SSN=ESSN AND
DNO=5 GROUP BY PNUMERO, PNAME;
```

Aqui restringimos as tuplas da relação (e daí as tuplas de cada grupo) àquelas que satisfazem a condição especificada na cláusula WHERE — a saber, aquelas dos que trabalham no departamento 5. Note que precisamos de cuidado extra quando são aplicadas duas condições diferentes (uma função na cláusula SELECT e outra na cláusula HAVING). Por exemplo, suponha que queiramos contar, em cada departamento, o número *total* de empregados cujos salários excedam os 40 mil dólares, mas somente para os departamentos onde houver mais de cinco empregados. Aqui, a condição (SALÁRIO>40000) aplica-se apenas à função COUNT da cláusula SELECT. Suponha que escrevamos a seguinte consulta *incorreta*:

```
SELECT          DNAME, COUNT (*)
FROM            DEPARTAMENTO, EMPREGADO
WHERE           DNUMERO=DNO AND SALARIO>40000
GROUP BY       DNAME
HAVING         COUNT (*) > 5;
```

Ela é incorreta porque vai selecionar somente os departamentos que tiverem mais de cinco empregados *que ganham mais de 40 mil dólares cada*. A regra é executar primeiro a cláusula WHERE, para selecionar as tuplas individualmente; a cláusula HAVING será aplicada depois, para selecionar grupos de tuplas individuais. Assim, as tuplas já estarão limitadas aos empregados que ganham mais de 40 mil dólares *antes* de a função da cláusula HAVING ser aplicada. Um jeito de formular corretamente essa consulta é usar uma consulta aninhada, como mostrado na Consulta 28.

Consulta 28

Para cada departamento que tenha mais de cinco empregados, recupere o número do departamento e o número dos empregados que recebem mais de 40 mil dólares.

```
Q28: SELECT      DNUMERO, COUNT (*)
FROM            DEPARTAMENTO, EMPREGADO WHERE DNUMERO=DNO AND
SALARIO>40000 AND DNO IN (SELECT      DNO
FROM            EMPREGADO GROUP BY DNO HAVING COUNT (*) > 5)
GROUP BY       DNUMERO;
```

8.5.9 Discussão e Resumo das Consultas SQL

Uma consulta em SQL pode conter mais de seis cláusulas, porém, somente as duas primeiras são obrigatórias — SELECT e FROM. As cláusulas são especificadas na seguinte ordem, e as cláusulas entre colchetes [...] são opcionais:

SELECT <ATRIBUTOS E LISTA DE FUNCOES>

FROM <LISTA DE TABELAS>

[WHERE <CONDICAO>]

[GROUP BY <ATRIBUTO(S) AGRUPADO(S)>]

[HAVING <CONDICAO DE AGRUPAMENTO>]

[ORDER BY <LISTA DE ATRIBUTOS>];

A cláusula SELECT relaciona os atributos ou as funções que serão recuperados. A cláusula FROM especifica todas as relações (tabelas) necessárias à consulta, incluindo as relações que serão compostas pela junção, mas não as usadas em consultas aninhadas.

A cláusula WHERE especifica as condições para a seleção de tuplas das relações, incluindo as condições para as junções, se necessário. A cláusula GROUP BY estabelece os atributos de agrupamento, embora a cláusula HAVING especifique uma condição sobre os grupos que estão sendo selecionados em vez de condições sobre as tuplas individualmente. As funções construtoras das funções agregadas COUNT, SUM, MIN, MAX e AVG são utilizadas em conjunto com o agrupamento, ainda que também possam ser aplicadas sobre todas as tuplas selecionadas em uma consulta sem a cláusula GROUP BY. Finalmente, ORDER BY especifica uma ordenação para a apresentação do resultado da consulta.

Uma consulta se desenrola, *conceitualmente*, primeiro pela aplicação da cláusula FROM (para identificar todas as tabelas envolvidas na consulta ou para concretizar alguma junção de tabelas), seguida da cláusula WHERE, então pelas cláusulas GROUP BY e HAVING. Conceitualmente, ORDER BY é aplicada no final para mesclar o resultado da consulta. Se nenhuma das três últimas cláusulas (GROUP BY, HAVING e ORDER BY) forem especificadas, poderemos pensar *conceitualmente* que a consulta estará sendo executada como segue: Para *cada combinação de tuplas* — cada uma proveniente de uma das relações determinadas na cláusula FROM — se desenvolve a cláusula WHERE; se o resultado for TRUE, serão colocados os valores dos atributos especificados na cláusula SELECT dessa combinação de tuplas no resultado da consulta. Naturalmente, essa não é uma maneira eficiente para a implementação da consulta em um sistema real, e cada SGBD possui rotinas especiais de otimização de consultas para determinar um plano de execução que seja eficiente. Vamos discutir o processamento e a otimização de consultas nos capítulos 15 e 16.

Em geral, uma mesma consulta SQL pode ser especificada de muitas maneiras diferentes. Essa flexibilidade na especificação de consultas possui vantagens e desvantagens. A principal vantagem é que o usuário pode escolher a técnica com a qual se sente mais confortável. Por exemplo, muitas consultas podem ser determinadas pelas condições de junção dentro da cláusula WHERE, ou usando junção de relações na cláusula FROM, ou por intermédio de alguma forma de consulta aninhada com o operador de comparação IN. Alguns usuários podem se sentir mais confortáveis com uma abordagem, enquanto outros preferem outra. Do ponto de vista do programador e do sistema, considerando-se a otimização de consultas, geralmente é preferível formular uma consulta com o mínimo de aninhamentos e ordenações possíveis.

A desvantagem de haver inúmeras formas de especificar uma mesma consulta é confundir o usuário, que pode não saber que técnica usar para formular um determinado tipo de consulta. Outro problema é que pode ser mais eficiente executar uma consulta de um jeito que de outro. O ideal seria não depender desses modos de formulação: o SGBD deveria processar a mesma consulta do mesmo jeito, independentemente do modo como fosse formulado. Mas isso é muito difícil na prática, uma vez que o SGBD possui diferentes métodos para processar consultas diferentes. Assim, o usuário tem como responsabilidade adicional determinar qual formulação é mais eficiente. O ideal seria que sua única preocupação fosse formular a consulta corretamente. Seria responsabilidade do SGBD executar a consulta de maneira eficiente. Na prática, no entanto, é de grande ajuda se o usuário souber quais tipos de construtores tornam o processamento de uma consulta mais trabalhoso que outros (Capítulo 16).

12 A forma real de ordenação de uma consulta depende da implementação; essa é apenas uma forma de visão conceitual da consulta de modo a formulá-la corretamente.

8.6 COMANDOS INSERT (INSERÇÃO), DELETE (EXCLUSÃO) E UPDATE (ATUALIZAÇÃO) EM SQL

Em SQL podem ser usados três comandos para modificar o banco de dados: INSERT, DELETE e UPDATE. Discutiremos cada um deles.

8.6.1. O comando INSERT

Em sua forma mais simples, o INSERT é usado para adicionar uma única tupla em uma relação. Devemos estabelecer o nome da relação e uma lista de valores para a tupla. Os valores devem ser relacionados *na mesma ordem* em que foram especificados os atributos correspondentes no comando CREATE TABLE. Por exemplo, para adicionar uma nova tupla à relação EMPREGADO da Figura 5.5, formulada pelo comando CREATE TABLE EMPREGADO da Figura 8.1, podemos usar U1:

U1: INSERT INTO EMPREGADO

VALUES ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak Forest,Katy,TX', 'M', 37000, '987654321', 4);

Uma segunda forma do comando INSERT permite ao usuário especificar explicitamente os nomes dos atributos que receberão os valores fornecidos por esse comando. Essa maneira é útil se uma relação possuir muitos atributos, mas somente para alguns poucos serão designados valores na nova tupla. Entretanto, os valores (*values* da cláusula INSERT) deverão incluir todos os atributos com especificação NOT NULL e sem o valor *default*. Os atributos com NULL permitido ou os valores DEFAULT serão aqueles que podem ser *deixados de fora*. Por exemplo, para inserir uma tupla de um novo EMPREGADO para o qual sabemos apenas os atributos PNAME, UNAME, DNO e SSN, podemos usar U1A:

U1A: INSERT INTO EMPREGADO (PNAME, UNAME, DNO, SSN) VALUES
('Richard', 'Marini', 4, '653298653');

Os atributos que não foram especificados em U1 A serão registrados com seus valores DEFAULT ou NULL, e os valores são relacionados na mesma ordem que os *atributos forem listados no próprio comando INSERT*. Também é possível inserir em uma relação *diversas tuplas*, separadas por vírgulas, em um único comando INSERT. Os valores relativos de *cada tupla* serão colocados entre parênteses.

Um SGBD que implemente a SQL-99 deve dar suporte e forçar todas as restrições de integridade que podem ser utilizadas na DDL. Entretanto, alguns SGBDs não incorporam todas as restrições, como forma de manter sua eficiência, e por causa da complexidade exigida para garantir todas as restrições. Se um sistema não dá suporte a alguma restrição — digamos, a de integridade referencial —, os usuários ou os programadores precisam garantir essa restrição. Por exemplo, se executarmos o comando U2 para o banco de dados mostrado na Figura 5.6, um SGBD que não desse suporte à integridade referencial faria a inserção mesmo não havendo nenhuma tupla DEPARTAMENTO no banco de dados com DNUMERO = 2. É responsabilidade do usuário checar se existe alguma restrição violada, uma vez *que essa verificação não é implementada no SGBD*. No entanto, os SGBDs deveriam implementar as verificações de modo a garantir todas as restrições de integridade *previstas na SQL*. Um SGBD que implementa NOT NULL vai rejeitar um comando INSERT, no qual um atributo declarado como NOT NULL não tenha nenhum valor; por exemplo, U2A poderia ser *rejeitado* porque o valor do SSN não foi informado:

U2: INSERT INTO EMPREGADO (PNAME, UNAME, SSN, DNO) VALUES
('Robert', 'Hatcher', '980760540', 2); (* U2 seria rejeitado se o SGBD implementasse a integridade referencial *)

U2A: INSERT INTO EMPREGADO (PNAME, UNAME, DNO) VALUES
('Robert', 'Hatcher', 5); (* U2A seria rejeitado se houvesse a verificação para impedir o registro de valores *null* pelo SGBD*)

Uma variação do comando INSERT é a inserção de diversas tuplas em uma relação por meio da conjunção da inserção do *resultado de uma consulta* em uma nova relação, que será criada. Por exemplo, para criar uma tabela temporária que tenha o nome de 'número de empregados' e que contenha o total de salários de cada departamento, podemos formular o comando U3A e U3B:

U3A: **CREATETABLE** DEPTS INFO

U3B:

```
(DEPTJMOME      VARCHAR(15),
NO_DE_EMPS      INTEGER,
TOTAL_SAL       INTEGER);
INSERT INTO    DEPTSJNFO (DEPT_NOME, NO_DE_EMPS
TOTAL_SAL)

SELECT         DNAME, COUNT (*), SUM (SALÁRIO)
FROM           (DEPARTAMENTO JOIN EMPREGADO ON
DNUMERO=DNO)

GROUP BY      DNAME;
```

A tabela DEPTSJNFO será criada por U3A e carregada com as informações resumidas recuperadas do banco de dados pela consulta U3B. Poderemos, assim, consultar DEPTSJNFO como qualquer outra relação; quando não precisarmos mais dela, poderemos removê-la usando o comando **DROP TABLE**. Observe que a tabela DEPTSJNFO não poderá ser atualizada, isto é, se atualizarmos as tabelas DEPARTAMENTO ou EMPREGADO depois de emitir o U3B, as informações de DEPTSJNFO *permanecerão desatualizadas*. Temos de criar uma visão (*view*), (Seção 9.2) para manter essa tabela atualizada.

8.6.2 O Comando DELETE

O comando **DELETE** remove tuplas de uma relação. Se incluir a cláusula **WHERE**, similar à que é usada nas consultas SQL, serão selecionadas as tuplas que serão deletadas. As tuplas serão explicitamente removidas de uma única tabela de cada vez. Entretanto, as remoções poderão propagar-se nas tuplas de outras relações, se forem definidas, em DDL, *ações engatilhadas* (*referential triggered actions*) nas restrições de integridade (Seção 8.2.2). Dependendo do número de tuplas selecionadas na condição da cláusula **WHERE**, poderão ser excluídas nenhuma, uma ou várias tuplas em um único comando **DELETE**. A omissão da cláusula **WHERE** determina que todas as tuplas da relação serão excluídas; entretanto, a definição da tabela, vazia, permanecerá no banco de dados. Os comandos **DELETE** em U4A a U4D, se aplicados independentemente no banco de dados da Figura 5.6, vão excluir, respectivamente, zero, uma, quatro e todas as tuplas da relação EMPREGADO:

U4A: **DELETE FROM**

WHERE U4B: DELETE FROM

WHERE U4C: DELETE FROM

WHERE

U4D: **DELETE FROM**

```
EMPREGADO      UNOME= 'Brown' ;      EMPREGADO      SSN= '123456789' ;
EMPREGADO DNO IN (SELECT FROM WHERE EMPREGADO ;
DNUMERO
DEPARTAMENTO
DNAME='Pesquisa');
```

8.6.3 O Comando UPDATE

O comando **UPDATE** é usado para modificar os valores dos atributos de uma ou mais tuplas. Como o comando **DELETE**, a cláusula **WHERE** em um comando **UPDATE** seleciona as tuplas de uma única relação que serão modificadas. Entretanto, uma atualização no valor da chave primária pode propagar-se para os valores das chaves estrangeiras, nas tuplas de outras relações, se essa *ação engatilhada* for especificada em DDL nas restrições de integridade referencial (Seção 8.2.2). Uma cláusula adicional **SET**, dentro do comando **UPDATE**, especifica os atributos que serão modificados e seus novos valores. Por exemplo, para alterar a localização e o número do departamento que controla o projeto número 10 para, respectivamente, 'Bellaire' e 5, usaremos U5:

U5: **UPDATE PROJETO**

SET PLOCALIZACAO = 'Bellaire', DNUM = 5

WHERE PNUMERO=10;

13 Outras ações poderão ser aplicadas automaticamente por meio dos gatilhos (Seção 24.1) e de outros mecanismos.

14 Devemos usar o comando **DROP TABLE** para remover a definição da tabela (Seção 8.3.1).

178

Capítulo 8 SQL-99: Definição de Esquema, Restrições Básicas e Consultas (*Queries*)

Um único comando UPDATE pode modificar várias tuplas. Um exemplo disso é dar um aumento de salário de 10% a todos os empregados do departamento 'Pesquisa', como mostrado na U6. Nessa formulação, a modificação do valor SALÁRIO depende do valor original do SALÁRIO de cada tupla, assim, serão necessárias duas referências ao atributo SALÁRIO. Na cláusula SET, a referência para o atributo SALÁRIO à direita refere-se ao valor antigo de SALÁRIO, *antes da modificação*, e à esquerda, ao novo valor do SALÁRIO, *após a modificação*:

U6: UPDATE EMPREGADO

SET SALÁRIO = SALÁRIO * 1.1

WHERE DNO IN (SELECT DNUMERO

FROM DEPARTAMENTO **WHERE** DNOME='Pesquisa');

Também é possível especificar o novo valor de um atributo como NULL ou DEFAULT. Observe que cada comando UPDATE refere-se explicitamente a somente uma relação. Para modificar as diversas relações, precisaremos emitir diversos comandos UPDATE.

8.7 OUTRAS FUNCIONALIDADES DA SQL

A SQL possui funcionalidades adicionais que não descreveremos neste capítulo, mas que discutiremos em outras partes deste livro. São elas:

- A SQL tem capacidade de especificar as restrições genéricas, chamadas asserções, usando o comando CREATE ASSERTION, que é descrito na Seção 9.1.
- A SQL possui construtores de linguagem para a especificação de visões (*views*), também conhecidas como tabelas virtuais, usando o comando CREATE VIEW. As visões são derivadas de tabelas básicas declaradas por meio do comando CREATE TABLE e são discutidas na Seção 9.2.
- A SQL possui técnicas diferentes para a criação de programas em várias linguagens de programação que podem incluir os comandos SQL para acessar um ou mais bancos de dados. Estas possuem SQL embutida (e dinâmica), SQL/CLI (*Call Language Interface* — interface para a chamada de linguagem), e seu predecessor ODBC (*Open Data Base Connectivity* — conectividade para o banco de dados aberta) e SQL/PSM (*Program Stored Modules* — módulos de programas armazenados). Discutiremos as diferenças entre essas técnicas na Seção 9.3, e então cada uma das técnicas entre as seções 9.4 e 9.6. Também discutiremos como acessar o banco de dados SQL pela linguagem de programação Java usando JDBC e SQLJ.
- Cada SGBDR comercial tem, além dos comandos SQL, um conjunto de comandos para a especificação dos parâmetros de projeto do banco de dados físico, bem como as estruturas de arquivos para as relações e os caminhos de acesso, como índices. Chamamos esses comandos, no Capítulo 2, de *linguagem para a definição de armazenamento* (*storage de-finition language* — SDL). As primeiras versões da SQL possuíam comandos para a **criação de índices**, mas estes foram removidos da linguagem porque não estavam no nível do esquema conceitual (Capítulo 2).
- A SQL possui comandos para o controle de transações. Eles são usados para especificar as unidades de processamento com o propósito de controle de concorrência e recuperação do banco de dados. Discutiremos esses comandos no Capítulo 17, após discutirmos os conceitos de transações com mais detalhes.
- A SQL possui construtores de linguagem para a especificação de *concessão e revogação de privilégios* aos usuários. Os privilégios normalmente correspondem a ter direito ao uso de determinados comandos SQL para acesso a certas relações. Para cada relação é designado um proprietário, e o proprietário ou o DBA pode conceder a um grupo de usuários o privilégio de usar um comando SQL — como SELECT, INSERT, DELETE ou UPDATE — para acessar uma relação. Adicionalmente, o DBA pode conceder privilégios para a criação de esquemas, tabelas ou visões para determinados usuários. Esses comandos SQL — chamados GRANT ou REVOKE — serão discutidos no Capítulo 23, no qual será tratada a questão da segurança e da autoridade de banco de dados.
- A SQL possui construtores de linguagem para a criação de gatilhos (*triggers*). Estes são, em geral, referidos como técnicas para **banco de dados ativo**, uma vez que especificam as ações que são automaticamente disparadas quando um evento, como as atualizações, ocorre no banco de dados. Discutiremos essas funcionalidades na Seção 24.1, na qual serão abordados os conceitos de banco de dados ativo.

- A SQL vem abrangendo diversas funcionalidades dos modelos orientados a objeto a fim de incorporar os recursos mais poderosos, adicionando vantagens aos sistemas relacionais conhecidos como **objeto-relacional**. Essas vantagens são a criação de atributos com estruturas mais complexas (também chamadas **relações aninhadas** — *nested relations*), especificando os tipos de dados abstratos (chamados UDTs ou *user-defined types* — tipos de dados definidos pelo usuário) para os atributos e as tabelas, criando **identificadores de objeto** para referência às tuplas e especificando as **operações** em tipos, discutidos no Capítulo 22.
- A SQL e o banco de dados relacional podem interagir com novas tecnologias, como XML (*eXtended Markup Language*; Capítulo 26) e OLAP (*On Line Analytical Processing for Data Warehouses*; Capítulo 28).

8.8 RESUMO

Neste capítulo apresentamos a linguagem de banco de dados SQL. Essa linguagem, ou suas variações, tem sido implementada como interface em diversos SGBDs comerciais, incluindo Oracle, DB2 e SQL/DS da IBM, SQL Server e ACCESS da Microsoft, INGRES, INFORMIX e SYBASE. A versão original da SQL foi implementada em um SGBD chamado SYSTEM R, que foi desenvolvido pela IBM Research. A SQL foi projetada para ser uma linguagem abrangente, incluindo as declarações para a definição de dados, consultas, visões e especificação de restrições. Discutimos várias dessas declarações em seções específicas deste capítulo. No final desta seção, debatemos as funcionalidades adicionais que serão descritas ao longo deste livro. Nossa ênfase foi o padrão SQL-99.

A Tabela 8.2 resume a sintaxe (ou a estrutura) de várias declarações SQL. Essa síntese não pretende ser completa nem descrever todos os construtores SQL possíveis; ela serve para uma consulta rápida para a maioria dos construtores disponíveis em SQL. Usamos a notação BNF, na qual os símbolos que não são fixos são mostrados entre os sinais <...>, as partes opcionais são mostradas entre colchetes [...], as repetições estão entre chaves (...) e as alternativas, entre parênteses (... I... I...).

TABELA 8.2 Resumo da Sintaxe SQL

```
CREATE TABLE <nome da tabela> (<nome da coluna> <tipo da coluna> [<restricoes de atributos>]
{,<nome da coluna> <tipo da coluna> [<restricoes de atributos>]} [<restricoes de tabelas> {,<restricoes de tabelas>}])
DROP TABLE <nome da tabela>
ALTER TABLE <nome da tabela> ADD <nome da coluna> <tipo da coluna>
SELECT [DISTINCT] <lista de atributos>
FROM (<nome da tabela> {<alias>} | <tabelas em juncao>) {, (<nome da tabela> {<alias>} | <tabelas em juncao>}
[WHERE <condicoes>]
[GROUP BY <atributos de agrupamento> [HAVING <condicao para a seleção de grupos>]]
[ORDER BY <nome da coluna> [<ordenacao>] {, <nome da coluna> [<ordenacao>]}]
<lista de atributos>::=(*|(<nome da coluna> | <funcao> (((DISTINCT) <nome da coluna> | *)))
{, (<nome da coluna> | <funcao> (((DISTINCT) <nome da coluna> | * )))) <atributos de agrupamento>::= <nome da coluna>
{, <nome da coluna>} <ordenacao>::= (ASC | DESC)
INSERT INTO <nome da tabela> [(<nome da coluna>{, <nome da coluna>})]
(VALUEs (<valores constantes>, {<valores constantes>}) {, (<valores constantes>{,<valores constantes>})}) |
<declaracao SELECT>
DELETE FROM <nome da tabela>
[ WHERE <condicoes de selecao> ]
UPDATE <nome da tabela>
SET <nome da coluna> = <expressao de valores> (, <nome da coluna> = <expressao de valores>}
```

15 A sintaxe completa da SQL-99 é descrita em documentos volumosos com centenas de páginas.

180 Capítulo 8 SQL-99: Definição de Esquema, Restrições Básicas e Consultas (*Queries*)

[WHERE<condicao de selecao>]

CREATE [UNIQUE] INDEX <nome do indice>

ON <nome da tabela> (<nome da coluna> [<ordenacao> {, (<nome da coluna> [<ordenacao>])}]

[CLUSTER]

DROP INDEX <nome do indice>

CREATE VIEW <nome da visao> [(<nome da coluna> {, <nome da coluna> })] AS <declaração SELECT>

DROP VIEW <nome da visao>

*Os dois últimos comandos não fazem parte do padrão SQL2.

Questões para Revisão

8.1. Como as relações (tabelas) em SQL diferem das relações definidas formalmente no Capítulo 5 ? Discuta outras diferenças na terminologia. Por que a SQL não permite tuplas repetidas em uma tabela ou no resultado de uma consulta?

8.2. Relacione os tipos de dados que são permitidos como atributos SQL.

8.3. Como a SQL viabiliza a implementação das restrições de integridade de entidade e referencial, descritas no Capítulo 5? E as ações referenciais engatilhadas (*triggered actions*)!

8.4. Descreva as seis cláusulas da sintaxe das consultas SQL e mostre quais os tipos de construtores que podem ser especificados em cada uma das seis. Quais cláusulas são obrigatórias e quais são opcionais?

8.5. Descreva conceitualmente como uma consulta SQL pode ser executada, especificando a ordem conceitual para a execução de cada uma das seis cláusulas.

8.6. Discuta como os NULLs são tratados em operadores de comparação na SQL. Como são tratados os NULLs quando funções agregadas são aplicadas em uma consulta em SQL? Como os NULLs são tratados se existirem em atributos agrupados?

Exercícios

8.7. Considere o banco de dados mostrado na Figura 1.2 cujo esquema é mostrado na Figura 2.1. Quais são as restrições de integridade referencial que poderiam ser obtidas nesse esquema? Escreva as declarações DDL SQL para a definição do banco de dados.

8.8. Repita o Exercício 8.7, mas usando o banco de dados do esquema COMPANHIA AÉREA da Figura 5.8.

8.9. Considere o esquema do banco de dados relacional BIBLIOTECA da Figura 6.12. Escolha a ação apropriada (*reject*, *cascade*, *set to null*, *set to default*) para cada restrição de integridade referencial, tanto para a *exclusão* da tupla quanto para a *atualização* de um valor do atributo-chave primário da tupla referida. Justifique suas escolhas.

8.10. Escreva as declarações DDL SQL apropriadas para definir o esquema do banco de dados relacional BIBLIOTECA da Figura 6.12. Especifique as chaves e as ações referenciais engatilhadas apropriadas.

8.11. Escreva as consultas SQL para aquelas do banco de dados BIBLIOTECA do Exercício 6.18.

8.12. Como podem ser garantidas as restrições de chave e de chave estrangeira pelo SGBD? Essa técnica que você sugeriu é difícil de implementar? A verificação da restrição pode ser executada eficientemente quando são aplicadas as atualizações no banco de dados?

8.13. Especifique as consultas do Exercício 6.16 em SQL. Mostre o resultado de cada consulta se ela fosse aplicada ao banco de dados EMPRESA da Figura 5.6.

8.14- Especifique as seguintes consultas adicionais ao banco de dados da Figura 5.5, em SQL. Mostre os resultados das consultas se cada uma fosse aplicada ao banco de dados da Figura 5.6.

a. Para cada departamento cuja média de salários dos empregados for maior que 30 mil dólares, recupere o nome desse departamento e o número de empregados que trabalham nesse departamento.

b. Suponha que queiramos saber o número dos empregados do sexo masculino em cada departamento, em vez de todos os empregados (como no Exercício 8.14a). Poderemos expressar essa consulta em SQL? Por que sim ou por que não ?

8.15. Especifique as atualizações para os Exercícios 5.10 usando os comandos de atualização SQL.

8.16. Especifique as seguintes consultas em SQL para o banco de dados da Figura 1.2:

a. Recupere os nomes de todos os estudantes veteranos graduados em *CC' (ciências da computação).

- b. Recupere os nomes de todos os cursos em que o professor King deu aulas entre 1998 e 1999.
 - c. Para cada disciplina do professor King, recupere o número do curso, o semestre, o ano e o número de estudantes dessa disciplina.
 - d. Recupere o nome e o histórico escolar de cada estudante veterano (Turma = 5) graduando em CC. Um histórico escolar compreende o nome do curso, seu número, créditos, semestre, ano e a nota obtida em cada curso completo do estudante.
 - e. Recupere os nomes e o departamento dos estudantes classe A (estudantes que têm nota A em todos os cursos realizados).
 - f. Recupere os nomes e o departamento de todos os estudantes que não obtiveram uma nota A em nenhum dos cursos que realizaram.
- 8.17. Escreva declarações de atualização SQL para fazer as seguintes modificações no esquema do banco de dados mostrado na Figura 1.2:
- a. Insira o novo estudante <'Johnson', 25, 1, 'MAT'>, no banco de dados.
 - b. Altere a classe do aluno 'Smith' para 2.
 - c. Insira o novo curso, <'Engenharia do Conhecimento', 'CC4390', 3, 'CC'>.
 - d. Apague o registro do aluno cujo nome seja 'Smith' e cujo número seja 17.
- 8.18. Especifique as consultas e as atualizações dos Exercícios 6.17 e 5.11, os quais se referem ao banco de dados COMPANHIA AÉREA (Figura 5.8), em SQL.
- 8.19. a. Faça um projeto de um esquema de banco de dados relacional para uma aplicação de banco de dados pessoal.
- b. Declare suas relações usando DDL SQL.
 - c. Especifique algumas consultas em SQL que serão necessárias para sua aplicação de banco de dados.
 - d. Com base nas expectativas de uso do seu banco de dados, escolha alguns dos atributos para a indexação.
 - e. Implemente seu banco de dados se você tiver um SGBD que dê suporte à SQL.
- 8.20. Especifique as respostas para os Exercícios 6.19 ao 6.21 e para o Exercício 6.23 em SQL.

Bibliografia Seleccionada

A linguagem SQL, denominada originalmente SEQUEL, teve por base a linguagem SQUARE (*Specifying Queries as Relational Expressions*), descrita por Boyce *et al.* (1975). A sintaxe da SQUARE foi modificada em SEQUEL (Chamberlin e Boyce, 1974) e então em SEQUEL 2 (Chamberlin *et al.* 1976), na qual a SQL é baseada. A implementação da SEQUEL foi feita pela IBM Research em San José, Califórnia.

Reisner (1977) descreve os fatores humanos na evolução da SEQUEL, na qual ela acha que os usuários têm algumas dificuldades para a especificação correta das condições de junção e de agrupamento. Date (1984b) faz uma crítica à linguagem SQL, ressaltando suas potencialidades e imperfeições. Date e Darwen (1993) descrevem a SQL2. A ANSI (1986) delineou o padrão original SQL e a ANSI (1992) descreveu o padrão SQL2. Vários vendedores de manuais descrevem as características da SQL como a que foi implementada no DB2, SQL/DS, Oracle, INGRES, INFORMIX e outros produtos comerciais de SGBD. Melton e Simon (1993) têm um tratamento abrangente da SQL2. Horowitz (1992) discute alguns problemas relativos à integridade referencial e à propagação de atualizações na SQL2.

A questão das atualizações de uma visão é, entre outros, tratada por Dayal e Bernstein (1978), Keller (1982) e Langerak (1990). As implementações de visões são discutidas em Blakeley *et al.* (1989). Negri *et al.* (1991) descrevem a semântica formal das consultas SQL.

9

Mais SQL: Asserções (Assertions), Visões (Views) e Técnicas de Programação

Nos capítulos anteriores descrevemos alguns aspectos da linguagem SQL, o padrão para o banco de dados relacional. Descrevemos as declarações SQL para a definição de dados, modificação de esquemas, consultas e atualizações. Também descrevemos a forma pela qual as restrições, como as de chave e referencial, são especificadas. Neste capítulo apresentaremos alguns aspectos adicionais da SQL. Começaremos a Seção 9.1 descrevendo a declaração CREATE ASSERTION, que permite a determinação de restrições genéricas sobre o banco de dados. Na Seção 9.2 descreveremos as facilidades SQL para a definição de visões (*views*) do banco de dados. Estas também são chamadas *tabelas virtuais* ou *derivadas*, pois mostram ao usuário o que parece ser uma tabela; entretanto, suas informações são derivadas de tabelas previamente definidas.

As outras seções deste capítulo discutem as diversas técnicas de acesso a banco de dados por meio de programas. A maioria dos acessos a bancos de dados, em situações práticas, é feita por intermédio de programas de software que implementam as aplicações de um banco de dados. Esses softwares são normalmente desenvolvidos em linguagens de propósito geral como JAVA, COBOL ou C/C++. Relembrando a Seção 2.3.1: Quando uma declaração de um banco de dados é inserida em um programa, a linguagem de programação utilizada é chamada *linguagem hospedeira* (*host language*), enquanto a linguagem de um banco de dados — SQL, no nosso caso — é conhecida como *sublinguagem de dados*. Em alguns casos, as *linguagens especiais de programação de um banco de dados* são desenvolvidas especificamente para a implementação de aplicações de um banco de dados. Embora muitas dessas linguagens tenham sido desenvolvidas como protótipos de pesquisa, algumas linguagens notáveis de programação para o banco de dados acabaram tendo seu uso disseminado, como o PL/SQL (*Programming Language/SQL*) da ORACLE.

Começaremos nossa apresentação de programação para o banco de dados na Seção 9.3, com uma visão geral das diferentes técnicas desenvolvidas para o acesso a um banco de dados por meio de programas. Assim, na Seção 9.4, discutiremos as regras para embutir as declarações SQL em uma linguagem de programação de uso geral, normalmente conhecida por *SQL embutida*. Essa seção também discutirá rapidamente a *SQL dinâmica*, cujas consultas podem ser construídas dinamicamente em tempo de execução, e apresenta as bases da SQLJ, uma variação da SQL embutida que foi desenvolvida especificamente para a linguagem de programação JAVA. Na Seção 9.5 discutiremos as técnicas conhecidas por *SQL/CLI* (*Call Level Interface* — interface em nível de chamada), na qual uma biblioteca de procedimentos e funções proporciona acesso ao banco de dados. Foram propostos vários conjuntos de funções para essa biblioteca. O conjunto de funções da SQL/CLI é fornecido pelo padrão SQL. A ODBC (*Open Data Base Connectivity* — conectividade aberta de um banco de dados) é outra biblioteca de funções. Não descreveremos a ODBC aqui, pois ela é considerada predecessora da SQL/CLI. A terceira biblioteca de funções — que descreveremos — é a JDBC, desenvolvida especialmente para o acesso a banco de dados por meio de JAVA. Finalmente, na Seção 9.6, discutiremos a *SQL/PSM* (*Persistem Stored Modules* — módulos persistentes), que é uma parte do padrão SQL que permite o armazenamento persistente de módulos — procedimentos e funções — de programas pelo SGBD, acessados por meio de SQL. A Seção 9.7 resume o capítulo.

9.1 ESPECIFICANDO AS RESTRIÇÕES GENÉRICAS POR ASSERÇÕES

Em SQL, os usuários podem especificar as restrições genéricas — que não se enquadram em nenhuma das categorias descritas na Seção 8.2 — via **asserções declarativas**, usando a declaração CREATE ASSERTION da DDL. Em uma asserção, é dado um nome à restrição por meio de uma condição semelhante à cláusula WHERE de uma consulta SQL. Por exemplo, para especificar a restrição 'O salário de um empregado não pode ser maior que o salário do gerente do departamento em que ele trabalha' em SQL, precisaremos formular a seguinte asserção:

```
CREATE ASSERTION LIMITE_SALARIO CHECK (NOT EXISTS (SELECT *  
FROM EMPREGADO E, EMPREGADO M, DEPARTAMENTO D WHERE E.SALARIO>M.SALÁRIO AND  
E.DNO=D.DNUMERO AND D.SSNGER=M.SSN) );
```

A restrição LIMITE_SALARIO é seguida da palavra-chave CHECK, que é seguida da **condição** entre parênteses que precisa ser verdadeira em todos os estados do banco de dados para que a asserção seja satisfeita. O nome da restrição pode ser usado posteriormente para sua referência, modificação ou eliminação. O SGBD é responsável por garantir que a condição não seja violada. Pode ser usada qualquer condição na cláusula WHERE, mas muitas condições podem ser especificadas utilizando-se o estilo EXISTS e NOT EXISTS das condições SQL. Sempre que alguma tupla no banco de dados fizer com que uma condição ASSERTION evolua para FALSE, a restrição será **violada**. A restrição será **satisfeita** para um dado estado do banco de dados, se nenhuma *combinação de tuplas* no banco de dados violar essa restrição.

A técnica básica para a formulação de asserções é especificar uma consulta que selecione as tuplas que *violam a condição desejada*. Por meio da inclusão dessa consulta em uma cláusula NOT EXISTS, a asserção especificará que o resultado dessa consulta deverá ser vazio. Logo, a asserção será violada se o resultado da consulta não for vazio. Em nosso exemplo, a consulta seleciona todos os empregados cujos salários sejam maiores que o do gerente de seu departamento. Se o resultado dessa consulta não for vazio, a asserção será violada.

Observe que a cláusula CHECK e a condição da restrição podem ser usadas também para especificar as restrições nos atributos e nos domínios (Seção 8.2.1) e nas tuplas (Seção 8.2.4). A maior diferença entre uma CREATE ASSERTION e as outras duas é que a cláusula CHECK da SQL em um atributo, um domínio ou uma tupla fará a checagem *somente quando as tuplas forem inseridas ou atualizadas*. Assim, a verificação poderá ser implementada com maior eficiência pelo SGBD nesses casos. O projetista do esquema poderia usar CHECK em atributos, nos domínios e nas tuplas apenas quando estiver certo de que a restrição *só poderá ser violada pela inserção ou atualização de tuplas*. Porém, o projetista deveria usar CREATE ASSERTION somente nos casos em que não for possível usar CHECK nos atributos, nos domínios ou nas tuplas, assim a verificação será implementada com maior eficiência pelo SGBD.

Outra declaração relacionada ao CREATE ASSERTION na SQL é a CREATE TRIGGER, mas os gatilhos serão usados de modo diferente. Em muitos casos é conveniente especificar o tipo de ação que deverá ser tomado quando certos eventos ocorrerem ou quando certas condições forem satisfeitas. Em vez de oferecer aos usuários somente a opção para a interrupção de uma operação que cause violação — como pela CREATE ASSERTION —, o SGBD poderia dispor de outras opções. Por exemplo, pode ser útil especificar uma condição em que, quando houver uma violação, os outros usuários sejam informados. Um gerente gostaria de ser informado, por meio do envio de uma mensagem, se as despesas de um empregado em viagem excedessem certo limite. A ação que o SGBD poderia tomar, nesse caso, é enviar a mensagem adequada ao usuário em questão. A condição será, assim, usada para **monitorar** o banco de dados. Outras ações poderiam ser especificadas, como executar um determinado procedimento armazenado (*stored procedure*) ou engatilhar outras atualizações. A declaração CREATE TRIGGER, na SQL, é usada para implementar essas ações. Um **gatilho** (trigger) especifica um **evento** (como uma operação de atualização no banco de dados em particular), uma **condição** e uma **ação**. A ação será executada automaticamente se uma condição for satisfeita quando ocorrer um determinado evento. Vamos abordar os gatilhos em detalhes na Seção 24.1 quando descrevermos o *banco de dados ativo*.

9.2 VISÕES (VIEWS — TABELAS VIRTUAIS) EM SQL

Nesta seção introduziremos o conceito de visão em SQL. Mostraremos como as visões são especificadas, discutiremos o problema das atualizações das visões e como uma visão pode ser implementada em um SGBD.

9.2.1 Conceito de Visão em SQL

Uma **visão**, na terminologia SQL, é uma tabela única derivada de outra tabela, que pode ser uma tabela básica ou uma visão previamente definida. Uma visão não existe de forma física, ela é considerada uma **tabela virtual**, em contraste com as tabelas básicas, cujas tuplas são realmente armazenadas no banco de dados. Isso limita as operações de atualização possíveis para as visões, embora não imponha nenhuma limitação para as consultas.

Podemos imaginar uma visão como um meio para a especificação de uma tabela que precise ser consultada freqüentemente, embora ela não exista fisicamente. Por exemplo, na Figura 5.5, podemos precisar de consultas freqüentes que recuperem os nomes dos empregados e os projetos nos quais eles trabalham. Em vez de formular uma junção entre as tabelas EMPREGADO, TRABALHA_EM e PROJETO toda vez que isso for necessário, podemos definir uma visão cujo resultado seja essa junção. Poderemos, então, consultar essa visão, que estará definida como uma única tabela, em vez de termos de fazer duas junções envolvendo as três tabelas para a recuperação desses dados. Chamaremos EMPREGADO, TRABALHA_EM e PROJETO de **tabelas de definição** da visão.

9.2.2 Especificação de Visões em SQL

Em SQL, o comando para especificar uma visão é o **CREATE VIEW**. A visão recebe um nome (virtual) de tabela (ou nome da visão), uma lista de nomes de atributos e uma consulta para especificar o conteúdo dessa visão. Se nenhum dos atributos da visão for resultado de uma função ou de uma operação aritmética, não teremos de especificar os nomes de atributos para a visão, dessa forma, eles receberiam como padrão os mesmos nomes que os atributos das tabelas de definição. As visões V1 e V2 criam tabelas virtuais cujos esquemas estão ilustrados na Figura 9.1 quando aplicados ao esquema de banco de dados da Figura 5.5.

```
V1: CREATE VIEW    TRABALHA_EM1
AS SELECT    PNAME, UNOME, PJNOME, HORAS
FROM        EMPREGADO, PROJETO, TRABALHA_EM
WHERE        SSN=ESSN AND PNO=PNUMERO;
V2: CREATE VIEW    DEPT_INFO(DEPT_NOME,NO_EMPS,TOTAL_SAL)
AS SELECT    DNAME, COUNT (*), SUM (SALÁRIO)
FROM        DEPARTAMENTO, EMPREGADO
WHERE        DNUMERO=DNO GROUP BY DNAME;
TRABALHA,  EM1
```

PNAME	UNOME	PJNOME	HORAS
-------	-------	--------	-------

DEPTJINFO

DEPT_NOME	NO_EMPS	TOTAL_SAL

FIGURA 9.1 Duas visões especificadas para o esquema do banco de dados da Figura 5.5

Em V1, não especificamos nenhum novo nome de atributo para a visão TRABALHA_EM1 (embora pudéssemos fazê-lo); nesse caso, TRABALHA_EM1 *receberá* os nomes dos atributos das tabelas de definição EMPREGADO, PROJETO e TRABALHA_EM. A visão V2 explicita novos nomes para os atributos da visão DEPTJINFO, usando a correspondência um a um entre os atributos especificados da cláusula **CREATE VIEW** e os especificados na cláusula **SELECT** da consulta que define a visão.

Poderemos agora especificar as consultas SQL em uma visão — ou tabela virtual —, do mesmo modo que se especificam as consultas envolvendo as tabelas básicas. Por exemplo, para recuperar o último e o primeiro nome de todos os empregados que trabalham no 'ProjetoX', poderemos utilizar a visão TRABALHA_EM1 e formular a consulta QV1:

```
QV1: SELECT PNAME, UNOME FROM TRABALHA_EM1 WHERE PJNOME='ProjetoX';
```

Na SQL, o termo *visão* é mais limitado que o termo *visão do usuário*, discutido nos capítulos 1 e 2, uma vez que a visão do usuário poderia possibilitar a inclusão de muitas relações.

9.2 Visões (Views — Tabelas Virtuais) em SQL 185

A mesma consulta poderia exigir a formulação de duas junções de relações básicas; uma das principais vantagens das visões é simplificar a especificação de certas consultas. As visões também são usadas para os mecanismos de autorização e segurança (Capítulo 23).

Supõe-se que uma visão esteja *sempre atualizada*; se modificarmos as tuplas das tabelas básicas sobre as quais a visão foi construída, a visão deverá, automaticamente, refletir essas alterações. Conseqüentemente, a visão não é realizada no instante de sua *definição*, mas quando *especificarmos uma consulta* sobre ela. É responsabilidade do SGBD, e não do usuário, ter a certeza de que uma visão está atualizada.

Se não precisarmos mais de uma visão, poderemos usar o comando DROP VIEW para dispensá-la. Por exemplo, para nos livrarmos da visão V1, podemos usar a declaração SQL V1A:

V1 A: DROP VIEW TRABALHA_EM1;

9.2.3 Implementação e Atualização da Visão

As implementações eficientes de visões para as consultas são um problema complexo. Existem duas abordagens principais. Uma estratégia, chamada modificação da consulta, implica modificar uma consulta de visão em uma consulta de tabelas básicas. Por exemplo, a consulta QV1 poderia ser automaticamente transformada pelo SGBD na seguinte consulta:

SELECT PNAME, UNAME

FROM EMPREGADO, PROJETO, TRABALHA_EM **WHERE** SSN=ESSN **AND** PNO=PNUMERO **AND** PJNOME='ProjetoX';

A desvantagem dessa abordagem é que ela é ineficiente para as visões definidas via consultas complexas que tenham a execução demorada, precisamente no caso de serem aplicadas diversas consultas à visão dentro de um curto espaço de tempo. Outra estratégia, chamada materialização da visão, implica criar fisicamente uma tabela temporária a partir da primeira consulta a essa visão e mantê-la, considerando que poderão seguir-se outras consultas. Nesse caso, deve-se ter uma estratégia eficiente para a atualização da tabela da visão sempre que as tabelas básicas forem atualizadas, de modo a garantir que a visão esteja sempre atualizada. Técnicas usando os conceitos de atualização incremental foram desenvolvidas para esse fim; elas determinam quais novas tuplas precisariam ser inseridas, deletadas ou modificadas na tabela materializada da visão, quando uma alteração for aplicada a uma das tabelas básicas de definição. Geralmente a visão é mantida o tempo necessário para que seja consultada. Se a visão não for consultada por um certo período, o sistema poderá automaticamente remover a tabela física e reconstruí-la, a partir do zero, quando novas consultas fizerem referência à visão.

As atualizações de visões são complicadas e podem ser ambíguas. Em geral, a atualização de uma visão definida em uma *única tabela, sem funções agregadas*, pode ser mapeada na atualização de uma tabela básica subjacente sob certas condições. Para uma visão envolvendo as junções, uma operação de atualização pode ser mapeada de *diversas maneiras*, por meio de operações de atualização nas tabelas básicas subjacentes. Para ilustrar o potencial de problemas resultantes de atualizações em uma visão definida sobre várias tabelas, considere a visão TRABALHA_EM1 e suponha que seja emitido um comando de atualização do atributo PJNOME de 'ProdutoX' para 'ProdutoY', e deste para 'John Smith'. Essa atualização da visão é mostrada em UV1:

UV1: UPDATE TRABALHA_EM1

SET PJNOME = 'ProdutoY'

WHERE UNOME='Smith' **AND** PNAME='John' **AND** PJNOME='ProdutoX';

Essa consulta pode ser mapeada em diversas atualizações das relações básicas para chegar ao efeito desejado pela atualização da visão. Duas atualizações possíveis, (a) e (b), nas relações básicas relativas a UV1, são mostradas aqui:

(a): UPDATE TRABALHA_EM

SET PNO= (SELECT PNUMERO

FROM PROJETO

WHERE PJNOME='ProdutoY')

WHERE ESSNIN (SELECT SSN

FROM EMPREGADO

WHERE UNOME='Smith' **AND** PNAME='John')

AND**PNO = (SELECT PNUMERO****FROM PROJETO WHERE PJNOME='ProdutoX'); (b): UPDATE PROJETO SET PJNOME = 'ProdutoY'****WHERE PJNOME = 'ProdutoX';**

A atualização (a) altera a tupla de 'John Smith' em PROJETO de 'ProdutoX' para 'ProdutoY', e essa alteração é a que provavelmente foi desejada. Entretanto, (b) também poderia dar o efeito desejado pela atualização da visão, embora o faça por intermédio da mudança do 'ProdutoX', na relação PROJETO, para 'ProdutoY'. É bastante improvável que o usuário que formulou a visão UV1 quisesse que a atualização fosse interpretada como (b), uma vez que isso teria efeito de alteração em todas as tuplas da visão com PJNOME= 'ProdutoX'.

Algumas atualizações de visões podem não fazer muito sentido; por exemplo, modificar o atributo TOTAL_SAL da visão DEPTJNFO não faz sentido, porque TOTAL_SAL é definido como a soma dos salários de todos os empregados. Essa solicitação é mostrada em UV2:

UV2: UPDATE DEPTJNFO**SET TOTAL_SAL=100000****WHERE DNOME='Pesquisa';**

Um grande número de atualizações nas relações básicas subjacentes pode satisfazer essa atualização de visão.

Uma atualização de visão é viável quando apenas *uma atualização possível*, nas relações básicas, puder realizar o efeito desejado por ela. Sempre que uma atualização em uma visão puder ser mapeada para *mais de uma atualização* nas relações básicas subjacentes, precisaremos ter certos cuidados para escolher a atualização desejada. Alguns pesquisadores desenvolveram métodos para a escolha das atualizações mais prováveis, ao passo que outros preferem obter essa escolha do usuário, para o mapeamento da atualização desejada na definição da visão.

Em resumo, podemos fazer as seguintes observações:

- Uma visão de uma única tabela de definição é atualizável se a visão contiver, entre seus atributos, a chave primária da relação básica, bem como todos os atributos com restrição NOT NULL *que não contiverem* valores *default* especificados.
- As visões definidas a partir de diversas tabelas utilizando-se as junções, em geral, não são atualizáveis.
- As visões definidas usando-se as funções de agrupamento e agregadas não são atualizáveis.

Na SQL, a cláusula WITH CHECK OPTION precisa ser adicionada no final da definição da visão se ela *puder ser atualizada*. Isso permite que o sistema cheque a capacidade de atualização da visão e planeje a estratégia de execução das atualizações das visões.

9.3 PROGRAMAÇÃO COM O BANCO DE DADOS: ESCOLHAS E TÉCNICAS

Vamos agora voltar nossa atenção para as técnicas desenvolvidas para acessar o banco de dados por meio de programas e, em particular, enfocar como ter acesso a um banco de dados SQL pelos programas aplicativos. Nossa apresentação da SQL, até então, enfocou os construtores para as diversas operações sobre o banco de dados — da definição do esquema e da especificação de restrições nas consultas, até atualizações e especificação de visões. A maioria dos sistemas de banco de dados possui interface interativa, na qual esses comandos SQL podem ser digitados diretamente em um monitor e enviados ao sistema de um banco de dados. Por exemplo, em um sistema computacional que tiver o ORACLE SGBDR instalado, o comando SQLPLUS vai iniciar a interface interativa. O usuário poderá digitar comandos SQL ou consultas diretamente, em algumas linhas finalizadas por ponto e vírgula e pela tecla Enter (isto é, "; <cr>"). Alternativamente, pode ser criado um arquivo de comandos, executado por uma interface interativa, digitando @<nome do arquivo>. O sistema vai executar os comandos do arquivo e apresentar seus resultados, se houver algum.

Uma interface interativa é muito conveniente para a criação de esquemas e restrições ou para consultas *ad hoc* eventuais. Entretanto, a maioria das interações com o banco de dados na prática é executada por meio de programas que tenham sido cuidadosamente projetados e testados. Esses programas são normalmente conhecidos como programas de aplicação ou aplicações com o banco de dados, e são usados como *transações customizadas* pelos usuários finais, conforme discutido na Seção 1.4.3. Outro uso muito comum da programação com o banco de dados é acessá-lo por meio de um programa de aplicação que implementa uma interface Web; por exemplo, para as reservas de passagens aéreas ou para compras em uma loja. De fato, a maioria das aplicações na Web em comércio eletrônico inclui algum comando de acesso a um banco de dados.

9.3 Programação com o Banco de Dados: Escolhas e Técnicas 187

Nesta seção daremos primeiramente uma visão geral das principais abordagens para a programação com o banco de dados. Então discutiremos alguns problemas que ocorrem quando tentamos acessar um banco de dados a partir de uma linguagem de programação (*general-purpose programming language*) e as seqüências típicas de comandos para a interação com um banco de dados a partir de um programa de software.

9.3.1 Abordagens para a Programação com o Banco de Dados

Existem diversas técnicas para as interações entre um banco de dados e os programas de aplicação. As principais abordagens para a programação com o banco de dados são as seguintes:

1. *Embutindo os comandos de banco de dados em uma linguagem de programação de propósito geral:* nessa abordagem, as declarações para o banco de dados ficam **embutidas** na linguagem de programação hospedeira, e elas são identificadas por um prefixo especial. Por exemplo, o prefixo embutido para a SQL é a cadeia EXEC SQL, que precede todos os comandos SQL na linguagem hospedeira. Um **pré-compilador** ou **pré-processador** inspeciona primeiro o código-fonte do programa para identificar as declarações do banco de dados e extraí-los para o processamento pelo SGBD. Eles serão recolocados no programa como chamadas de funções para o gerador de código do SGBD.
2. *Usando uma biblioteca de funções para o banco de dados:* deixa-se uma **biblioteca de funções** disponível para que a linguagem de programação hospedeira possa fazer chamadas para o banco de dados. Por exemplo, pode haver funções para conectar o banco de dados, fazer uma consulta, executar uma atualização, e assim por diante. A consulta real ao banco de dados e os comandos de atualização, bem como outras informações necessárias, são inseridos como parâmetros nas chamadas de funções. Essa abordagem proporciona o que é conhecido como uma **Interface para o Programa de Aplicação** (*Application Programming Interface* — API) pelo programa de aplicação para o banco de dados.
3. *Projetando uma nova linguagem:* uma **linguagem de programação de um banco de dados** é projetada especialmente para ser compatível com o modelo do banco de dados e com a linguagem de consulta. As estruturas de programação adicionais, como laços (*loops*) e declarações condicionais, são acrescentadas à linguagem de um banco de dados para convertê-la em uma linguagem de programação completa.

Na prática, as duas primeiras abordagens são mais comuns, uma vez que as muitas aplicações existentes em linguagens de programação genéricas exigem algum acesso a um banco de dados. A terceira abordagem é a mais apropriada para as aplicações que tenham interação intensa com o banco de dados. Um dos problemas principais das duas primeiras abordagens é a *impedância de correspondência*, que não ocorre na terceira. Esse assunto será discutido a seguir.

9.3.2 Impedância de Correspondência (*Impedance Mismatch*)

A **impedância de correspondência** (*impedance mismatch*) é o termo usado para se referir aos problemas que ocorrem em decorrência das diferenças entre os modelos de um banco de dados e da linguagem de programação. Por exemplo, o modelo relacional possui três construtores principais: os atributos e seus tipos de dados, as tuplas (registros) e as tabelas (conjuntos de diversos registros). O primeiro problema possível é que os tipos de dados da linguagem de programação difiram dos tipos de dados do modelo de dados. Daí será necessário estabelecer uma forma de correspondência para cada linguagem de programação hospedeira que determine, para cada tipo de atributo, os tipos compatíveis na linguagem de programação. É necessário determinar **paridade** para cada linguagem de programação porque cada uma possui seus próprios tipos de dados; por exemplo, os tipos de dados disponíveis em C e JAVA são diferentes e ambos diferem dos tipos de dados da SQL.

Outro problema é que o resultado da maioria das consultas é um conjunto (ou diversos conjuntos) de tuplas e cada uma delas é formada por uma seqüência de valores de atributos. Em alguns programas, freqüentemente é necessário acessar o valor de um dado em particular, dentro de uma determinada tupla, para a impressão ou o processamento. Assim, os valores de paridade são necessários para mapear a *estrutura de dados resultante da consulta*, que é uma tabela apropriada para a estrutura de dados na linguagem de programação. É necessário um mecanismo de laço para varrer as tuplas do resultado de uma consulta, de forma a ter acesso a uma única tupla de cada vez e dela extrair os valores em particular. Um **cursor** ou uma **variável iterativa** (*iterator variable*) é usado para varrer as tuplas do resultado de uma consulta. Os valores individuais dentro de cada tupla são, em geral, extraídos por meio de variáveis, com o tipo apropriado dentro do programa.

2 Outros prefixos algumas vezes são usados, mas este é o mais comum.

* Lembramos que *iteração* significa iterar, tornar a fazer, repetir, enquanto *interação* significa ação de interagir, ação recíproca, ação mutua. (N. deT.)

A impedância de correspondência é minimizada quando uma linguagem de programação é especialmente projetada para usar os mesmos modelos de dados e os mesmos tipos de dados do modelo do banco de dados. Um exemplo disso é a PL/SQL da ORACLE. Para banco de dados de objetos, o modelo de objetos (Capítulo 20) é muito similar ao modelo de dados da linguagem de programação JAVA, assim a impedância de correspondência é amplamente reduzida quando JAVA é usada como linguagem hospedeira para o acesso a um banco de dados de objetos JAVA-compatíveis. Diversas linguagens de programação foram implementadas como protótipos de pesquisa (notas bibliográficas).

9.3.3 Seqüência Típica de Interação em Programação com o Banco de Dados

Quando um programador ou um engenheiro de software cria um programa que exige acesso a um banco de dados, é muito comum que o programa rode em um computador enquanto o banco de dados está instalado em outro. Lembre-se de que, na Seção 2.5, uma das arquiteturas comuns para o acesso a um banco de dados é o modelo cliente/servidor, no qual um **programa cliente** trabalha a lógica do software aplicativo, mas insere algumas chamadas a um ou mais **servidores** de banco de **dados** para o acesso ou a atualização de dados. Quando codificamos esse tipo de programa, uma seqüência de interação comum é a seguinte:

1. Quando um programa cliente exige acesso a um banco de dados em particular, o programa precisa primeiro *estabelecer* ou *abrir* uma **conexão** com o servidor de banco de dados. Normalmente isso envolve a especificação de um endereço na Internet (URL) de uma máquina na qual o servidor do banco de dados está localizado, além do fornecimento do *login* de uma conta e uma senha de acesso ao banco de dados.

2. Uma vez estabelecida a conexão, o programa pode interagir com o banco de dados submetendo-o a consultas, atualizações e outros comandos. Em geral, a maioria das declarações SQL pode ser inserida nos programas aplicativos.

3. Quando o programa não precisar mais do acesso ao banco de dados em questão, pode *terminar* ou *fechar* a conexão.

Um programa pode acessar diversos bancos de dados se necessário. Em algumas abordagens de programação com o banco de dados, somente uma conexão pode ser ativada por vez, embora, em outras, diversas conexões possam ser estabelecidas ao mesmo tempo.

Nas próximas três seções, discutiremos os exemplos de cada uma das três abordagens para a programação com o banco de dados. A Seção 9.4 descreve como a SQL é *embutida* em uma linguagem de programação. A Seção 9.5 discute como *as chamadas de funções* são usadas para o acesso ao banco de dados, e a Seção 9.6 trata de uma extensão da SQL, chamada SQL/PSM, que incorpora certos construtores das *linguagens de propósito geral*, utilizados para a definição de módulos (procedimentos e funções) que são armazenados dentro do sistema de banco de dados.

9.4 SQL EMBUTIDA, SQL DINÂMICA E SQLJ

9.4.1 Recuperando as Tuplas Isoladas com a SQL Embutida

Nesta seção daremos uma visão geral de como as declarações SQL podem ser embutidas em uma linguagem de programação de propósito geral como C, ADA, COBOL ou PASCAL. A linguagem de programação é chamada **linguagem hospedeira** (ou **linguagem host**). A maioria das declarações SQL — incluindo a definição de dados ou de restrições, as consultas, as atualizações ou a definição de visões — pode ser embutida em uma linguagem de programação hospedeira. Uma declaração SQL embutida se distingue de uma declaração da linguagem pelas palavras-chave EXEC SQL, usadas como prefixo, de modo que o **pré-processador** (ou **pré-compilador**) poderá separar as declarações SQL embutidas do código da linguagem hospedeira. As declarações SQL podem terminar com um ponto e vírgula (;) ou até encontrar um END-EXEC.

Para ilustrar os conceitos de SQL embutida, vamos usar C como linguagem de programação hospedeira. Dentro dos comandos SQL embutidos, podemos nos referir especificamente a variáveis declaradas do programa C. Estas são chamadas **variáveis compartilhadas** porque são usadas em ambos os programas, no C e nos comandos embutidos. As variáveis compartilhadas têm como prefixo dois pontos (:) *quando elas aparecem em uma declaração SQL*. Isso distingue os nomes das variáveis dos nomes dos construtores do esquema do banco de dados, como os atributos e as relações. Também permite que as variáveis de programa

3 Como discutido na Seção 2.5, existem arquiteturas *two-tier and three-tier* (duas camadas e três camadas). Para simplificar nossa discussão, vamos admitir, aqui, a arquitetura cliente/servidor *two-tier*. Discutiremos as variações adicionais dessa arquitetura no Capítulo 25.

4 Embora a SQL/PSM não seja considerada uma linguagem de programação completa, isso ilustra como os construtores típicos das linguagens de programação comuns — como os laços (*loops*) e as estruturas de condição — podem ser incorporados à SQL.

9.4 SQL Embutida, SQL Dinâmica e SQLJ 189

tenham os mesmos nomes que os atributos, desde que eles sejam diferenciados, na declaração SQL, pelo prefixo ":". Os nomes dos construtores do esquema do banco de dados — como os atributos e as relações — podem ser usados apenas dentro de comandos SQL, mas as variáveis compartilhadas podem ser utilizadas em qualquer parte do programa C sem o prefixo ":".

Suponha que queiramos escrever programas C para processar o banco de dados EMPRESA da Figura 5.5. Precisaremos declarar as variáveis de modo compatível com os tipos dos atributos do banco de dados que o programa vai acessar. O programador poderá escolher os nomes das variáveis de programa, que podem ou não ter nomes idênticos para os atributos correspondentes. Vamos usar as variáveis no programa C declaradas na Figura 9.2 para todos os nossos exemplos, e também mostrar os segmentos do programa C sem a declaração das variáveis. As variáveis compartilhadas são declaradas dentro de uma seção de declaração (*declare section*) no programa, como mostrado na Figura 9.2 (linhas 1 a 7). Os poucos tipos compatíveis entre C e SQL são os seguintes. Os tipos SQL INTEGER, SMALLINT, REAL e DOUBLE são mapeados com os tipos C long, short, float e double, respectivamente. As cadeias de caracteres de tamanho fixo e de tamanho variável (CHAR[i], VARCHAR[i]) em SQL podem ser mapeadas por matrizes de caracteres (char[i+1], varchar[i+1]) em C, as quais são um caractere mais longas que os tipos SQL, uma vez que as cadeias em C são terminadas por um caractere "\0" (*null*), que não faz parte da cadeia de caracteres propriamente dita.

Observe que os comandos SQL embutidos na Figura 9.2 são somente as linhas 1 e 7, as quais dizem ao pré-compilador para tomar nota dos nomes das variáveis C definidas entre BEGIN DECLARE e END DECLARE porque elas poderão aparecer nas declarações SQL embutidas — nas quais estarão precedidas por dois pontos (:). As linhas 2 a 5 correspondem a declarações normais em C. As variáveis de programa declaradas nas linhas 2 a 5 correspondem aos atributos das tabelas EMPREGADO e DEPARTAMENTO do banco de dados EMPRESA da Figura 5.5 que foram declarados pela DDL SQL da Figura 8.1. As variáveis declaradas na linha 6 — SQLCODE e SQLSTATE — são usadas para a comunicação de erros e condições de exceção entre o sistema de um banco de dados e o programa. A linha 0 mostra uma variável de laço do programa que não será usada em nenhuma declaração SQL embutida; logo, está fora da seção de declaração SQL.

```
0) int loop;
1) EXEC SQL BEGIN DECLARE SECTION ;
2) varchar dnome [16], pnome [16], unome [16], endereço [31] ;
3) char ssn [10], datnasc [11], sexo [2], minicial [2] ;
4) float salário, aumento;
5) int dno, dnumero ;
6) int SQLCODE ; char SQLSTATE [6] ;
7) EXEC SQL END DECLARE SECTION ;
```

FIGURA 9.2 Variáveis do programa C usadas nos exemplos de SQL embutida E1 e E2.

Conectando o Banco de Dados. Os comandos SQL para o estabelecimento de uma conexão com um banco de dados têm a seguinte forma:

**CONNECT TO <nome do servidor> AS <nome da conexão AUTHORIZATION>
<nome e senha do usuário da conta> ;**

Em geral, desde que um usuário ou um programa possam acessar diversos servidores de banco de dados, várias conexões poderão ser estabelecidas, mas somente uma conexão poderá estar ativa por vez. O programador ou o usuário pode usar o <nome da conexão> para trocar a conexão ativa com o seguinte comando:

SET CONNECTION <nome da conexão>;

Uma vez que a conexão não seja mais necessária, ela poderá ser encerrada por meio do seguinte comando:

DISCONNECT <nome da conexão>;

Nos exemplos deste capítulo, pressupomos que a conexão apropriada já tenha sido estabelecida com o banco de dados EMPRESA e que ela seja a conexão ativa corrente.

5 Usamos a numeração da linha em nossos segmentos de código para facilitar a referência; esses números não fazem parte do código.

6 As cadeias de caracteres SQL também podem ser mapeadas para tipos char* em C.

190 Capítulo 9 Mais SQL: Asserções (Assertions), Visões (Views) e Técnicas de Programação Comunicações entre o Programa e o SGBD Usando SQLCODE e SQLSTATE. As duas variáveis de comunicação especiais que são usadas pelo SGBD para comunicar as condições de erro ou exceção para o programa são SQLCODE e SQLSTATE. A variável SQLCODE mostrada na Figura 9.2 é uma variável inteira. Depois que cada comando do banco de dados for executado, o SGBD devolve um valor para o SQLCODE. Um valor 0 indica que a declaração foi executada com sucesso pelo SGBD. Se SQLCODE > 0 (ou, mais especificamente, se SQLCODE = 100), isso indica que não há mais dados (registros) disponíveis no resultado da consulta. Se SQLCODE < 0, isso indica que ocorreu algum erro. Em certos sistemas — por exemplo, no SGBDR ORACLE —, o SQLCODE é um campo de um registro estruturado chamado SQLCA (SQL *communication area* — área de comunicação SQL), assim, ele é referido como SQLCA.SQLCODE. Nesse caso, a definição de SQLCA precisa ser incluída no programa C pela inserção da seguinte linha:

```
EXEC SQL include SQLCA ;
```

Nas últimas versões do padrão SQL, foi adicionada uma variável de comunicação chamada SQLSTATE, que é uma cadeia de cinco caracteres. O valor "00000" na SQLSTATE indica que não houve nenhum erro ou exceção; outros valores apontam a ocorrência de erros ou exceções. Por exemplo, SQLSTATE "02000" indica que 'não há mais dados'. Atualmente, tanto a SQLSTATE quanto a SQLCODE estão disponíveis no padrão SQL. Muitos dos códigos de erros e exceções assumidos pela SQLSTATE estão, teoricamente, padronizados entre os diversos vendedores e as plataformas SQL, enquanto os códigos de retorno SQLCODE não são padronizados, mas definidos pelos representantes dos SGBDs. Logo, em geral, é melhor usar a SQLSTATE, porque por meio dela é possível tratar os erros em programas aplicativos independentemente de um SGBD em particular. Como exercício, o leitor pode reescrever os exemplos dados adiante neste capítulo usando SQLSTATE em vez do SQLCODE.

Exemplo de Programação com a SQL Embutida. Nosso primeiro exemplo para ilustrar a programação com a SQL embutida é um segmento de repetição de programa (laço — *loop*) que lê o número do seguro social de um empregado e imprime algumas informações do registro do EMPREGADO correspondente no banco de dados. O código do programa C é mostrado no segmento de programa E1 da Figura 9.3. O programa lê (entrada) o valor do número social e então recupera, do banco de dados, a tupla EMPREGADO com o número do seguro social via comando SQL embutido. A cláusula INTO (linha 5) especifica as variáveis de programa nas quais os valores dos atributos do banco de dados foram recuperados. As variáveis do programa C na cláusula INTO são prefixadas com dois pontos (:), conforme discutimos anteriormente.

A linha 7, em E1, ilustra a comunicação entre o banco de dados e o programa por meio da variável especial SQLCODE. Se o valor devolvido pelo SGBD para a SQLCODE for 0, a declaração anterior foi executada sem ocorrência de erro ou exceção. Na linha 7 isso pode ser verificado e pressupõe-se que, caso tenha ocorrido erro, é porque não existe nenhuma tupla EMPREGADO com o valor do número do seguro social em questão; o que resulta na mensagem que contém essa informação (linha 8).

Em E1, é selecionada uma *única tupla* pela consulta SQL embutida; e somente assim poderemos associar esses valores de atributos diretamente às variáveis do programa C por meio de uma cláusula INTO na linha 5. Normalmente, uma consulta SQL pode recuperar muitas tuplas. Nesse caso, o programa C vai tratar e processar as tuplas recuperadas uma por vez. Um *cursor* é usado para permitir o processamento de uma tupla de cada vez dentro do programa na linguagem hospedeira. Descreveremos os cursores a seguir.

9.4.2 Recuperando Várias Tuplas com a SQL Embutida Usando Cursores

Podemos imaginar um cursor como um ponteiro que aponta para uma única *tupla (linha)* do resultado de uma consulta que retornou diversas tuplas. O cursor será declarado quando o comando da consulta SQL for declarado no programa. Adiante no programa, um comando OPEN CURSOR vai buscar (*fetches*) o resultado da consulta no banco de dados e vai colocar o cursor na posição *anterior à primeira linha* do resultado da consulta. Esta se torna a linha corrente para o cursor. Em seguida, o comando FETCH é declarado no programa; cada FETCH move o cursor para a *próxima linha* do resultado da consulta, fazendo dela a linha corrente e passando os valores de seus atributos para as variáveis do programa C (linguagem hospedeira) especificadas pelo comando FETCH por meio da cláusula INTO. A variável cursor é basicamente um repetidor que varre (laços) as tuplas do resultado de uma consulta — uma por vez. É similar ao tradicional um-registro-por-vez (*record-at-a-time*) no processamento de arquivos.

7 Em particular, os códigos SQLSTATE, que começam com os caracteres de 0 a 4 ou de A a H, teoricamente estão padronizados, ao passo que outros valores podem ser definidos na implementação.

9.4 SQL Embutida, SQL Dinâmica e SQLJ 191

```
//Segmento de Programa E1:
0)   loop = 1 ;
1)   while (loop) {
2)       prompt ("Entre com o Numero do Seguro Social: ", ssn) ;
3)       EXEC SQL
4)           select PNAME, MINICIAL, UNOME, ENDEREÇO, SALÁRIO
5)           into :pname, :minicial, :unome, :endereco, :salario
6)           from EMPREGADO where SSN = :ssn ;
7)       if (SQLCODE == 0) printf (pname, minit, unome, endereço, salário)
8)       else printf ("Numero do Seguro Social nao existe: ", ssn) ;
9)       prompt("Mais Números de Seguro Social (entre 1 para Sim, 0 para Nao):
", loop) ;
10)  }
```

FIGURA 9.3 Segmento de Programa E1, um segmento de programa C com a SQL embutida.

Para determinar quando todas as tuplas do resultado de uma consulta foram processadas, a variável de comunicação SQLCODE (ou, alternativamente, SQLSTATE) é checada. Se um comando FETCH emitido levar o cursor para além da última tupla do resultado da consulta, um valor positivo (SQLCODE>0) será devolvido para a SQLCODE, indicando que nenhum dado (tupla) foi encontrado (ou a cadeia "02000" é devolvida para a SQLSTATE). O programador usa esse recurso para finalizar os laços sobre as tuplas do resultado de uma consulta. Em geral, podem ser abertos diversos cursores ao mesmo tempo. Um comando CLOSE CURSOR é emitido para indicar que o processamento do resultado de uma consulta associado ao cursor em questão foi terminado.

Um exemplo do uso de cursores é apresentado na Figura 9.4, na qual um cursor chamado EMP é declarado na linha 4. Presumimos que as variáveis apropriadas foram declaradas no programa C, como na Figura 9.2. O segmento de programa E2 lê (entrada) o nome de um departamento (linha 0), recupera seu número de departamento (linhas 1 a 3), e então recupera os empregados que trabalham naquele departamento via um cursor. Daí um laço (linhas 10 a 18) varre todos os registros empregados, um por vez, e imprime o nome dos empregados. O programa lê, então, o aumento de salário daquele empregado (linha 12) e atualiza o banco de dados com o novo valor do salário do empregado no resultado (linhas 14 a 16).

//Segmento de Programa E2:

```
0)   prompt ("Entre com o Nome do Departamento: ", dnome) ;
1)   EXEC SQL
2)       select DNUMERO into :dnumero
3)       from DEPARTAMENTO where DNAME = :dnome ;
4)   EXEC SQL DECLARE EMP CURSOR FOR
5)       select SSN, PNAME, MINICIAL, UNOME, SALÁRIO
6)       from EMPREGADO where DNO = :dnumero
7)       FOR UPDATE OF SALÁRIO ;
8)   EXEC SQL OPEN EMP ;
9)   EXEC SQL FETCH from EMP into :ssn, :pname, :minicial, :unome, :salario ;
10)  while (SQLCODE < 0) {
11)      printf("Nome do empregado e:", pname, minit, unome)
12)      prompt("Entre com o aumento de salário: ", aumento) ;
13)      EXEC SQL
14)          update EMPREGADO
15)          set SALÁRIO = SALÁRIO + :raise
16)          where CURRENT OF EMP ;
17)      EXEC SQL FETCH from EMP into :ssn, :pname, :minicial, :unome,
:salario ;
18)  }
19) EXEC SQL CLOSE EMP ;
```

FIGURA 9.4 Segmento de programa E2, um segmento de programa C que usa cursor com a SQL embutida para propósito de atualização.

Quando um cursor é definido para linhas que serão modificadas (atualizações), precisaremos adicionar a cláusula FOR UPDATE OF à declaração do cursor e listar os nomes dos atributos que serão atualizados pelo programa. Isso é ilustrado na linha 7 do segmento de código E2. Caso alguma linha seja deletada, as palavras-chave FOR UPDATE precisam ser adicionadas, sem especificar quaisquer atributos. Em comandos embutidos de UPDATE (ou DELETE), a condição WHERE CURRENT OF <nome do cursor> especifica que a tupla que está sendo referida pelo cursor será atualizada (ou deletada), como na linha 16 de E2.

192 Capítulo 9 Mais SQL: Asserções (Assertions), Visões (Views) e Técnicas de Programação

Observe que a declaração de um cursor e sua associação a uma consulta (linhas 4 a 7 em E2) não executam a consulta; esta será executada apenas quando o comando OPEN <nome do cursor> (linha 8) for executado. Note também que não existe necessidade de inserir a cláusula FOR UPDATE OF na linha 7 de E2 se o resultado da consulta for usado *somente para propósito de recuperação* (não para atualização ou remoção).

Algumas opções podem ser especificadas na declaração do cursor. O formato geral de uma declaração de cursor é o seguinte:

```
DECLARE <nome do cursor> [ INSENSITIVE ] [ SCROLL ] CURSOR
[ WITH HOLD ] FOR <especificação da consulta>
[ ORDER BY <especificação da ordenação> ]
[ FOR READ ONLY I FOR UPDATE [ OF <lista de atributos> ] ] ;
```

Já discutimos rapidamente as opções relacionadas à última linha. O *default* é considerar a consulta com propósito de recuperação de dados (FOR READ ONLY). Se alguma das tuplas da consulta resultado for atualizada, precisaremos especificar FOR UPDATE OF <lista de atributos> e relacionar os atributos que podem ser atualizados. Se alguma das tuplas for deletada, precisaremos especificar FOR UPDATE sem a lista de atributos.

Quando a palavra-chave opcional SCROLL for especificada na declaração de um cursor, é possível posicioná-lo de outras maneiras além da ordem puramente seqüencial. Uma orientação de busca pode ser adicionada ao comando FETCH, e seus valores podem ser NEXT (próximo), PRIOR (anterior), FIRST (primeiro), LAST (último), ABSOLUTE *i* (posição absoluta *i*) e RELATIVE *i* (posição relativa *i*). Nos dois últimos comandos, *i* precisa ser um valor inteiro que especifique uma posição absoluta de uma tupla ou uma posição relativa de uma tupla em relação à posição corrente do cursor, respectivamente. A orientação de busca *default* que utilizamos em nossos exemplos é NEXT. Essa orientação de busca permite que o programador mova o cursor sobre as tuplas do resultado da consulta com maior flexibilidade, proporcionando acesso randômico para posicionamento ou acesso em ordem reversa. Quando SCROLL é especificado em um cursor, a forma geral do comando FETCH é a seguinte, com as partes opcionais entre colchetes:

```
FETCH [ [ <orientação da busca> ] FROM ] <nome do cursoo INTO <lista de busca
designada>;
```

A cláusula ORDER BY ordenará as tuplas, assim o comando FETCH as buscará na ordem estabelecida. Essa ordem pode ser especificada de modo similar pela cláusula correspondente nas consultas SQL (Seção 8.4.6). As duas últimas opções na declaração de um cursor (INSENSITIVE e WITH HOLD) referem-se às características de transações em programas de banco de dados, que serão discutidas no Capítulo 17.

9.4.3 Especificando as Consultas em Tempo de Execução Usando a SQL Dinâmica

Nos exemplos anteriores, as consultas SQL embutidas foram codificadas como parte do código-fonte de um programa hospedeiro. Daí, a qualquer momento que queiramos formular uma consulta diferente, precisaremos escrever um novo programa e seguir todos os passos envolvidos (compilação, correção, teste e assim por diante). Em alguns casos, é conveniente escrever um programa que possa executar consultas diferentes em SQL, ou atualizações (ou outras operações), *dinamicamente em tempo de execução*. Por exemplo, podemos desejar escrever um programa que aceite uma consulta SQL digitada em um monitor, em seguida executá-la e apresentar seu resultado, como as interfaces interativas disponíveis na maioria dos SGBDRs relacionais. Outro exemplo seria a criação de uma interface amigável para gerar consultas SQL dinâmicas do usuário, com base em operações disparadas pelo clique em um ponto de um esquema gráfico (por exemplo, tipo interface QBE; Apêndice D). Nesta seção, daremos uma visão rápida da SQL dinâmica, que é uma técnica para gerar esse tipo de programa para o banco de dados por meio de um exemplo simples, que ilustre como a SQL dinâmica pode trabalhar.

O segmento de programa E3 da Figura 9.5 lê uma cadeia de caracteres, que é inserida pelo usuário (essa cadeia poderia ser um comando de atualização SQL), em uma variável sql atualizacadeia, tipo cadeia de caractere, na linha 3. Assim se prepara então o comando SQL da linha 4, pela associação com a variável SQL sql comando. Então, a linha 5 executa o comando. Observe que, nesse caso, a sintaxe não é verificada, nem outro tipo de verificação no comando será possível em tempo *de compilação*, uma vez que o comando não está disponível até sua execução. Isso contrasta com nossos exemplos anteriores de SQL embutida, em que a consulta pode ser verificada em tempo de compilação porque seu texto é checado no código do programa-fonte.

Embora seja relativamente direto em SQL dinâmica incluir um comando para atualização dinâmica, uma consulta dinâmica é muito mais complicada. Isso porque, geralmente, não sabemos nem o tipo nem o número de atributos que serão recuperados pela consulta SQL quando se escreve o programa. Algumas vezes, serão necessárias estruturas complexas de dados

9.4 SQL Embutida, SQL Dinâmica e SQLJ 193

para acolher diferentes tipos e quantidades de atributos do resultado de uma consulta, caso não haja nenhuma informação anterior conhecida sobre a consulta dinâmica. Técnicas similares à que discutiremos na Seção 9.5 podem ser usadas para designar os resultados de consultas (e parâmetros de consultas) para as variáveis de programas hospedeiros.

Em E3, a razão para a separação entre PREPARE e EXECUTE é que, mesmo que um comando seja executado diversas vezes em um programa, ele precisará ser preparado apenas uma vez. A preparação de um programa geralmente implica a verificação, pelo sistema, da sintaxe, e outros tipos de verificações, bem como a geração do código para a sua execução. É possível combinar os comandos PREPARE e EXECUTE (linhas 4 e 5 em E3) em uma única declaração escrevendo

EXEC SQL EXECUTE IMMEDIATE :sqlatualizacadeia ;

Essa forma é adequada caso o comando seja executado somente uma vez. Alternativamente, poderão ser separados um do outro, de modo a capturar qualquer erro após a declaração PREPARE, se houver algum.

//Segmento de Programa E3:

```
0) EXEC SQL BEGIN DECLARE SECTION ;
1) varchar sqlatualizacadeia [256] ;
2) EXEC SQL END DECLARE SECTION ;
3) prompt ("Entre com o Comando de Atualização: ", sqlatualizacadeia) ;
4) EXEC SQL PREPARE sqlcomando FROM :sqlatualizacadeia ;
5) EXEC SQL EXECUTE sqlcomando ;
```

FIGURA 9.5 Segmento de Programa E3, um segmento de programa c que usa a SQL dinâmica para a atualização de uma tabela.

9.4.4 SQLJ: SQL Embutida em Comandos JAVA

Na seção anterior, demos uma visão geral de como os comandos SQL podem ser embutidos em linguagem de programação tradicional, utilizando, em nossos exemplos, a linguagem C. Vamos agora voltar nossa atenção para a forma de embutir a SQL em um programa com linguagem orientada a objeto, em particular a linguagem JAVA. A SQLJ é um padrão que tem sido adotado por alguns vendedores para embutir a SQL em JAVA. Historicamente, a SQLJ foi desenvolvida depois de JDBC, que é usado para acessar os bancos de dados com a SQL por meio de JAVA, usando chamadas de função. Discutiremos o JDBC na Seção 9.5.2. Em nossa discussão, focalizaremos a SQLJ da forma como ela é usada no SGBDR ORACLE. Geralmente, um tradutor SQLJ converte as declarações SQL em JAVA, as quais podem ser executadas, então, por uma interface JDBC. Conseqüentemente, é necessário instalar um *driver* JDBC para usar a SQLJ. Nesta seção, focalizaremos como usar conceitos de SQLJ para escrever a SQL embutida dentro de um programa JAVA.

Antes de poder processar a SQLJ com JAVA em ORACLE, é necessário importar várias bibliotecas de classe, mostradas na Figura 9.6. Elas incluem as classes JDBC e IO (linhas 1 e 2), mais as classes adicionais relacionadas nas linhas 3,4e 5. Além disso, o programa deve, primeiro, conectar-se ao banco de dados desejado, usando a chamada função getConnection, que é um dos métodos da classe oracle, linha 5, da Figura 9.6. O formato dessa chamada de função, que retorna a um objeto do tipo *default context* (contexto-padrão), é o seguinte:

```
public static DefaultContext
getConnection(String uri, String usuário, String senha, Boolean
autoCommit)
throws SQLException;
```

Podemos escrever, por exemplo, as declarações das linhas 6 a 8 da Figura 9.6 para conectar um banco de dados ORACLE, localizado na URL <nome url>, usando o *login* <nome do usuário> e <senha>, com efetivação automática de cada comando, e então estabelecer essa conexão como *default context* para os comandos subseqüentes.

8 Esta seção pressupõe a familiaridade com os conceitos de orientação a objeto e com os conceitos básicos de JAVA. Se essa familiaridade faltar aos leitores, deve-se adiar esta seção para depois da leitura do Capítulo 20.

9 Discutiremos os *drivers* de JDBC na Seção 9.5.2.

10 Um *default context* (*contexto default*), quando marcado, é aplicado aos comandos subseqüentes do programa até que seja alterado.

11 *Efetivação automática* significa que cada comando será aplicado ao banco de dados após sua execução. A alternativa seria que o programador desejasse executar vários comandos relacionados para submetê-los em conjunto. Discutiremos os conceitos de submissão no Capítulo 17, quando descreveremos as transações em banco de dados.

194 Capítulo 9 Mais SQL: Asserções (Assertions), Visões (Views) e Técnicas de Programação

```

1) import java.sql.* ;
2) import java.io.*;
3) import sqlj.runtime.*;
4) import sqlj.runtime.ref.* ;
5) import oracle.sqlj.runtime.* ;
6) DefaultContext cntxt =
7)     oracle.getConnection("<nome url>", "<nome usuario>", "<senha>", true) ;
8) DefaultContext.setDefaultContext(cntxt) ;

```

FIGURA 9.6 Importando as classes necessárias para inserir a SQLJ em um programa JAVA no ORACLE, estabelecendo uma conexão e *default context*.

Nos exemplos seguintes, não serão mostradas as classes nem os programas completos em JAVA, uma vez que não é nossa intenção ensinar aqui a linguagem JAVA. Assim, mostraremos segmentos de programas que ilustrem o uso de SQLJ. A Figura 9.7 apresenta as variáveis de programa JAVA usadas em nossos exemplos. O segmento de programa J1, da Figura 9.8, lê o número do seguro social de um empregado e imprime algumas das informações presentes no banco de dados.

Observe que, como o JAVA já usa o conceito de controlar exceções por erro, uma exceção especial, chamada *SQLException*, é usada para devolver condições de erro ou exceções após a execução de um comando SQL pelo banco de dados. Ela tem um papel semelhante à da *SQLCODE* e da *SQLSTATE*, da SQL embutida. O programa JAVA tem muitos tipos de exceções predefinidas. Cada operação (função) de JAVA tem de estabelecer as exceções que podem ser emitidas — isto é, as condições de exceção que podem ocorrer enquanto se executa o código JAVA daquela operação. Se uma determinada exceção acontecer, o sistema transfere o controle ao código JAVA especificado para tratar aquela exceção. Em J1, o controle de exceção para um *SQLException* é especificado nas linhas 7 e 8. As exceções que podem ser emitidas pelo código, em uma operação em particular, devem ser especificadas como parte da declaração da operação ou da *interface* — por exemplo, no seguinte formato:

<tipo de retorno da operacao> <nome da operacao>(<parametros>) throws *SQLException*, *IOException* ;

Em SQLJ, os comandos de SQL embutidos dentro de um programa JAVA são precedidos por *#sql*, como ilustrado na linha 3 de J1, assim poderão ser identificados pelo pré-processador. A SQLJ usa uma *cláusula INTO* — similar à usada em SQL embutida — para devolver os valores dos atributos recuperados do banco de dados por uma consulta SQL em variáveis de programa JAVA. Da mesma forma que na SQL embutida, as variáveis de programa são precedidas de dois pontos (:) na declaração SQL.

```

1) String dnome, ssn, pnome, fn, unome, ln, datanasc, endereco ;
2) Char sexo, minicial, mi ;
3) double salário, sal ;
4) Integer dno, dnumero ;

```

FIGURA 9.7 Variáveis de programa JAVA usadas nos exemplos SQLJ J1 e J2.

//Segmento de Programa J1:

```

1) ssn = readEntry("Entre com o Numero do Seguro Social: ") ;
2) try {
3)     #sql {select PNAME, MINICIAL, UNOME, ENDEREÇO, SALÁRIO
4)         into :pname, :minicial, :unome, :endereco, :salario
5)         from EMPREGADO where SSN = :ssn};
6) } catch (SQLException se) {
7)     System.out.println("Numero do Seguro Social Inexistente: " + ssn) ;
8)     Return ;
9) }
10) System.out.println(pnome + " " + minicial + " " + unome + " " + endereco + " " + salário)

```

FIGURA 9.8 Segmento de Programa J1, um segmento de programa JAVA com a SQLJ.

Em J1, é selecionada uma tupla isolada pela consulta SQLJ embutida; é por isso que podemos marcar os valores desses atributos diretamente nas variáveis do programa JAVA, da cláusula INTO, na linha 4. Para as consultas que recuperam muitas tuplas, a SQLJ usa o conceito de um *iterator*, semelhante ao de um cursor da SQL embutido.

9.4.5 Recuperando Diversas Tuplas em SQLJ por meio de Iteradores

Em SQLJ, um **iterador** é um tipo de objeto associado a uma coleção (um ou diversos conjuntos) de tuplas do resultado de uma consulta. O iterador está associado às tuplas e aos atributos que aparecerem no resultado de uma consulta. Há dois tipos de iteradores:

1. Um **iterador designado**, que é associado ao resultado de uma consulta por intermédio da lista de nomes e tipos dos atributos que nela aparecerem.

2. Um **iterador posicional**, que lista somente os tipos *dos atributos* que aparecem no resultado da consulta.

Em ambos os casos, a lista deve estar *na mesma ordem* em que os atributos estão relacionados na cláusula SELECT da consulta. Porém, como veremos adiante, o uso do laço sobre o resultado de uma consulta é diferente entre os dois tipos de iteradores. Primeiro, mostraremos um exemplo do uso de um *iterador designado* na Figura 9.9, com o segmento de programa J2A. A linha 9 da Figura 9.9 mostra como um iterador designado, tipo Emp, é declarado. Note que os nomes dos atributos em um iterador designado devem casar com os nomes dos atributos no resultado da consulta SQL. A linha 10 mostra como um objeto iterador e de tipo Emp é criado no programa e então associado a uma consulta (linhas 11 e 12).

Quando o objeto iterador é associado a uma consulta (linhas 11 e 12 da Figura 9.9), o programa vai buscar o resultado da consulta no banco de dados e coloca o iterador na posição *anterior à primeira linha* do resultado da consulta. Esta se torna a tupla corrente para o iterador. Na sequência, são emitidas as **próximas** operações para o iterador; cada uma delas movimenta o iterador para a *próxima linha* do resultado da consulta, tornando-a a linha corrente. Se a linha existir, a operação recupera, na variável correspondente do programa, o valor do atributo para aquela linha. Se não houver mais nenhuma linha, a próxima operação devolverá *null*, que pode ser usado para o controle do laço.

//Segmento de Programa J2A:

```
0) dnome = readEntry ("Entre com o Nome do Departamento: ") ;
D try {
2)      #sql {select DNUMERO into :dnumero
3)          from DEPARTAMENTO where DNome = :dnome} ;
4)      } catch (SQLException se) {
5)          System.out.println ("Departamento Inexistente: " + dnome) ;
6)          Return ;
7)      }
8)      System.out.println("Informacoes dos Empregados do Departamento: " + dnome) ;
9)      #sql iterator Emp(String ssn, String pname, String minicial, String unome, double salário) ;
10)     Emp e = null ;
11)     #sql e = {select ssn, pname, minicial, unome, salário
12)               from EMPREGADO where DNO = :dnumero} ;
13)     while (e.next()) {
14)         System.out.println (e.ssn + " " + e.pnome + " " + e.minicial + " " +
15)         e.unome + " " + e.salário) ;
16)     } ;
16) e.close();
```

FIGURA 9.9 Segmento de programa J2A, um segmento de programa JAVA que usa um iterador designado para a impressão das informações dos empregados de um departamento em particular.

Na Figura 9.9, o comando (e.next()) da linha 13 executa duas funções: adquire a próxima tupla do resultado da consulta e controla o laço do *while*. Uma vez terminado o resultado da consulta, o comando e.close() (da linha 16) fecha o iterador.

Agora, considere o mesmo exemplo usando iteradores *posicionais* como mostrado na Figura 9.10 (segmento de programa J2B). A Linha 9 da Figura 9.10 mostra como um iterador posicional tipo Emppos é declarado. A diferença principal entre ele e o iterador designado é que não há nenhum nome de atributo no iterador posicional — somente tipos de atributos. Eles ainda devem ser compatíveis com os tipos de atributos do resultado da consulta SQL, e na mesma ordem em que nela aparecem. A linha 10 mostra como uma variável iterador posicional e, de tipo Emppos, é criada no programa, e então associada a uma consulta (linhas 11 e 12).

12 Discutiremos iteradores com mais detalhes no Capítulo 21, quando discutirmos sobre o banco de dados orientado a objetos.

196 Capítulo 9 Mais SQL: Asserções (Assertions), Visões (Views) e Técnicas de Programação

Até certo ponto, o comportamento de um iterator posicional é semelhante à SQL embutida (Seção 9.4.2). É necessário um comando de busca (*fetch*) para <variável iterator> nas <variáveis do programa> para recuperar a próxima tupla do resultado da consulta. A primeira vez que a busca é executada, a primeira tupla (linha 13 na Figura 9.10) é trazida. A Linha 16 adquire a próxima tupla até que não existam mais tuplas no resultado da consulta. Para controlar o laço, é usado um iterator posicional com função *e.endFetch()*. Essa função é marcada inicialmente com um valor TRUE quando o iterator é associado a uma consulta SQL (linha 11) e é marcado com FALSE cada vez que um comando de busca retornar com uma tupla válida do resultado da consulta. É fixado com TRUE novamente quando um comando de busca não achar mais nenhuma tupla. A linha 14 mostra como os laços são controlados pela negação.

//Segmento de Programa J2B:

```
0) dnome = readEntry ("Entre com o Nome do Departamento: ") ;
1) try {
2)     #sql {select DNUMERO into :dnumero
3)         from DEPARTAMENTO where DNO = :dnome} ;
4) } catch (SQLException se) {
5)     System.out.println ("Departamento Inexistente: " + dnome) ;
6)     Return ;
7) }
8) System.out.println ("Informações dos Empregados do Departamento: " +
dnome) ;
9) #sql iterator Emppos (String, String, String, String, double) ;
10) Emppos e = null ;
11) #sql e = {select ssn, pnome, minicial, unome, salário
12)         from EMPREGADO where DNO = :dnumero} ;
13) #sql {fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
14) while (!e.endFetch()) {
15)     System.out.println (ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
16)     #sql {fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
17) } ;
18) e.close() ;
```

FIGURA 9.10 Segmento de programa J2B, um segmento de programa JAVA que usa um iterator posicional para imprimir as informações dos empregados de um departamento em particular.

9.5 PROGRAMAÇÃO EM UM BANCO DE DADOS POR MEIO DE FUNÇÕES (FUNCTIONS CALLS): SQL/CLI E JDBC

A SQL embutida (Seção 9.4) é referida, às vezes, como uma abordagem estática de programação com banco de dados, porque a formulação da consulta é inserida no programa e não pode ser alterada sem recompilação ou reprocessamento do código fonte. O uso de chamadas de função é mais dinâmico em um banco de dados que em um programa que embutiu a SQL. Já vimos uma técnica de programação dinâmica de um banco de dados — SQL dinâmica — na Seção 9.4.3. As técnicas discutidas aqui proporcionam outra abordagem para a programação dinâmica de um banco de dados. Uma biblioteca de funções, também conhecida como uma interface para a programação de aplicações (API), é usada para acessar o banco de dados. Embora isso proporcione maior flexibilidade, uma vez que não é necessário nenhum pré-processador, uma desvantagem é que a verificação da sintaxe, entre outras, nos comandos SQL, precisará ser feita em tempo de execução. Outra desvantagem é que, às vezes, é necessária uma programação mais complexa para acessar o resultado da consulta, porque pode ser que o número e os tipos dos atributos do resultado da consulta não sejam conhecidos antecipadamente.

Nesta seção faremos uma avaliação de duas interfaces de chamada de função. Discutiremos primeiro SQL/CLI (*Call Level Interface* — Interface em Nível de Chamada), que é parte do padrão da SQL. Foi desenvolvida em continuidade à técnica anterior, conhecida como ODBC (*Open Data Base Connectivity* — Conectividade Aberta para o Banco de Dados). Usamos C como linguagem hospedeira nos nossos exemplos de SQL/CLI. Daremos, então, uma avaliação de JDBC, que é uma interface de acesso a um banco de dados por chamada de função JAVA. Embora JDBC seja normalmente assumido como Java Data Base Connectivity (conectividade JAVA com o banco de dados), ela é apenas uma marca da Sun Microsystems, não um acrônimo.

9.5 Programação em um Banco de Dados por meio de Funções (Functions Calls): SQL/CLI e JDBC 197

A principal vantagem de usar uma interface por chamada de função é a maior facilidade de acesso a diversos bancos de dados dentro de um mesmo programa de aplicação, até mesmo se eles estiverem armazenados em SGBDs diferentes. Isso será discutido, a seguir, na Seção 9.5.2, quando veremos a programação JAVA em um banco de dados com JDBC, embora essa vantagem também se aplique à programação SQL/CLI e ODBC em um banco de dados (Seção 9.5.1).

9.5.1 Programação com o Banco de Dados por meio de SQL/CLI Usando C como Linguagem Hospedeira

Antes de se usar as chamadas de função SQL/CLI, é necessário instalar a biblioteca apropriada no servidor de um banco de dados. Esses pacotes são obtidos com o vendedor do SGBD que está sendo usado. Daremos uma avaliação de como SQL/CLI pode ser utilizada em um programa C. Ilustraremos nossa apresentação com o segmento de um programa exemplo CLI1 mostrado na Figura 9.11.

Ao usar SQL/CLI, as declarações de SQL são dinamicamente criadas e passadas como cadeias de parâmetros nas chamadas de função. Consequentemente, é necessário manter o controle das interações entre o programa hospedeiro e o banco de dados, em uma estrutura de dados em tempo de execução, porque os comandos de um banco de dados são processados em tempo de execução. A informação é mantida em quatro tipos de registros, representados em tipos de dados *structs* em C. Um **registro de ambiente** é usado como um recipiente (*container*) para manter as informações de controle sobre as conexões com o banco de dados e registrar as informações do ambiente. Um **registro de conexão** cria informações de controle da conexão de um banco de dados em particular. Um **registro de declaração** mantém o controle das informações necessárias a uma declaração SQL. Um **registro de descrição** conserva as informações sobre as tuplas ou os parâmetros — por exemplo, o número e os tipos dos atributos na tupla ou o número e os tipos dos parâmetros em uma chamada de função.

//Programa CLI 1:

```
0) #include sqlcli.h ;
1) void printSal () {
2)     SQLHSTMT stmtl ;
3)     SQLHDBC conl ;
4)     SQLHENV envl ;
5)     SQLRETURN retl, ret2, ret3, ret4 ;
6)     retl = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, Senvl) ;
7)     if (!retl) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, envl, Sconl) else exit ;
8)     if (!ret2) ret3 = SQLConnect(conl, "dbs", SQL_NTS, "js", SQL_NTS, "xyz", SQL_NTS) else exit ;
9)     if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, conl, &stmtl) else exit ;
10)    SQLPrepare (stmtl, "select UNOME, SALÁRIO from EMPREGADO where SSN = ?", SQL_NTS) ;
11)    prompt("Entre com o numero do seguro social: ", ssn) ;
12)    SQLBindParameter (stmtl, 1, SQL_CHAR, &ssn, 9, &fetchlenl) ;
13)    retl = SQLExecute(stmtl) ;
14)    if (!retl) {
15)        SQLBindCol (stmtl, 1, SQL_CHAR, Sunome, 15, &fetchlenl) ;
16)        SQLBindCol (stmtl, 2, SQL_FLOAT, &salario, 4, &fetchlen2) ;
17)        ret2 = SQLFetch (stmtl) ;
18)        if (!ret2) printf(ssn, unome, salário)
19)            else printf ("Numero do Seguro Social Inexistente: ", ssn) ;
20)    }
21) }
```

FIGURA 9.11 Segmento de programa CLI1, um segmento de programa C com SQL/CLI.

Cada registro torna-se acessível ao programa por uma variável C ponteiro — chamado **controle** (*handle*) de registro. O **ponteiro de controle** será devolvido quando um registro for criado. Para criar um registro e devolver seu ponteiro de controle, é usada a seguinte função de SQL/CLI:

SQLAllocHandle(<handle tipo>, <handle 1>, <handle 2>)

198 Capítulo 9 Mais SQL: Asserções (Assertions), Visões (Views) e Técnicas de Programação

Nessa função, os parâmetros são:

- `<handle e_tipo>` indica o tipo de registro que está sendo criado. Os valores possíveis para esse parâmetro são as palavras-chave `SQL_HANDLE_ENV`, `SQL_HANDLE_DBC`, `SQL_HANDLE_STMT` ou `SQL_HANDLE_DESC`, para um ambiente, uma conexão, uma declaração ou um registro de descrição, respectivamente.
- `<handle_l>` indica o recipiente (*container*) dentro do qual o novo controle está sendo criado. Por exemplo, para um registro de conexão, poderia ser o ambiente dentro do qual a conexão está sendo criada; para um registro de declaração, poderia ser a conexão para aquela declaração.
- `<handle_2>` é o ponteiro (*handle*) para o registro, do tipo `<handle_tipo>`, recém-criado.

Para codificar um programa em C que incluirá chamadas ao banco de dados por SQL/CLI, os passos normalmente seguidos estão descritos a seguir. Ilustraremos esses passos nos referindo ao exemplo CLI1, da Figura 9.11, que lê o número do seguro social de um empregado e imprime seu sobrenome e seu salário:

1. A *biblioteca de funções* que compreende SQL/CLI deve estar contida no programa C. Ela é chamada `sql cli.h` e é incluída pela linha 0 na Figura 9.11.
2. Devem ser especificadas no programa as *variáveis para os controles* necessários, tipo `SQLHSTMT`, `SQLHDBC`, `SQLHENV` e `SQLHDESC` para as declarações, as conexões, os ambientes e as descrições, respectivamente (linhas 2 a 4). Também devem ser declaradas as variáveis do tipo `SQLRETURN` (linha 5) para registro do código de retorno das chamadas de funções SQL/CLI. Um código de retorno 0 (zero) indica que a *execução* da chamada de função ocorreu com *sucesso*.
3. Um *registro de ambiente* deve ser marcado em um programa usando `SQLAllocHandle`. Para isso é usada a função exibida na linha 6. Como o registro de ambiente não está contido em nenhum outro registro no momento da criação do ambiente, o parâmetro `<handle_l>` é `null`, `SQL_NULL_HANDLE` (ponteiro nulo). O controle (ponteiro) para o registro de ambiente recém-criado é devolvido na variável `envl`, linha 6.
4. Um *registro de conexão* deve ser marcado em um programa utilizando `SQLAllocHandle`. Na linha 7, o registro de conexão criado tem `conl` como ponteiro de controle e está contido no ambiente `envl`. Uma conexão é então estabelecida em `conl` para um servidor de banco de dados em particular, usando a função `SQLConnect` da SQL/CLI (linha 8). Em nosso exemplo, o nome do servidor de um banco de dados que estamos conectando é o `'dbs'`, a conta e a senha para *login* são `'js'` e `'xyz'`, respectivamente.
5. Um *registro de declaração* deve ser marcado em um programa empregando `SQLAllocHandle`. Na linha 9, o registro de declaração criado tem `stml` como controle e usa o `conl` para conexão.
6. Uma declaração é *preparada* usando a função `SQLPrepare` da SQL/CLI. Na linha 10, é designada a cadeia (a consulta de nosso exemplo) SQL para o controle (*handle*) de declaração `stml`. O símbolo de ponto de interrogação (?) da linha 10 representa um parâmetro da declaração, que é um valor a ser determinado em tempo de execução — normalmente ligado a uma variável do programa C. Em geral, ocorrem vários parâmetros. Eles são diferenciados pela ordem de aparecimento dos pontos de interrogação dentro da declaração (o primeiro? representa o parâmetro 1; o segundo?, o parâmetro 2, e assim por diante). O último parâmetro da `SQLPrepare` deveria fornecer o comprimento da cadeia da declaração SQL em bytes, mas se entrarmos com a palavra-chave `SQL_NTS`, ela indicará que a cadeia que contém a consulta é terminada com um *null* (*null-terminated string* — *nts*, cadeia de caracteres terminada por nulo), para que a SQL possa calcular o comprimento da cadeia automaticamente. Isso também se aplica a outros parâmetros de cadeia nas chamadas de função.
7. Antes de executar a consulta, qualquer parâmetro deve ser ligado a variáveis de programas, usando a função SQL/CLI `SQLBindParameter`. Na Figura 9.11, o parâmetro (indicado por ?) da consulta preparada e referido por `stml` é ligado à variável `ssn` do programa de C na linha 12. Se houvesse *n* parâmetros na declaração SQL, deveria haver *n* chamadas de função `SQLBindParameter`, cada uma com um parâmetro de posição diferente (1, 2, ..., *n*).
8. Seguindo essas preparações, podemos então executar a declaração SQL controlada por `stml` utilizando a função `SQLExecute` (linha 13). Observe que, embora a consulta seja executada na linha 13, seus resultados não foram ainda associados a nenhuma variável do programa C.
9. Para determinar em que o resultado da consulta será devolvido, uma técnica comum é a das colunas associadas similares. Nela, cada coluna do resultado de uma consulta é associada a uma variável do programa C por meio da função `SQLBindCol`.

13 Não mostraremos aqui a descrição dos registros, de modo a manter uma apresentação simplificada.

9.5 Programação em um Banco de Dados por meio de Funções (Functions Calls): SQL/CLI e JDBC 199

As colunas são distinguidas pela ordem de seu aparecimento na consulta SQL. Na Figura 9.11, linhas 15 e 16, as duas colunas da consulta (UNOME e SALÁRIO) são ligadas às variáveis unome e salario, do programa C, respectivamente.

10. Finalmente, para recobrar os valores das colunas nas variáveis do programa C, é usada a função SQLFetch (linha 17). Essa função é semelhante ao comando FETCH (vá buscar) da SQL embutida. Se um resultado de consulta tiver uma coleção de tuplas, cada chamada SQLFetch entra com a próxima tupla e recupera seus valores de coluna nas variáveis de programa associadas. O SQLFetch devolve um código de exceção (diferente de zero) se não houver mais nenhuma tupla. Como pudemos ver, o uso de chamadas de função dinâmicas exige muita preparação para a montagem das declarações SQL e para associar os parâmetros do resultado da consulta às variáveis apropriadas do programa.

Na CLI1, uma única tupla é selecionada pela consulta SQL. A Figura 9.12 mostra um exemplo para a recuperação de diversas tuplas. Pressupomos que as variáveis apropriadas do programa C foram declaradas como na Figura 9.2. O segmento de programa CLI2 lê (entrada — input) o número de um departamento e, então, recupera os empregados que trabalham nesse departamento. Nesse caso, um laço itera os registros dos empregados, um a um, e imprime o último nome e o salário de cada um deles.

//Segmento de Programa CLI2:

```

0)  #include sqlcli.h ;
1)  void printDepartmentEmps() {
2)      SQLHSTMT stmt1 ;
3)      SQLHDBC con1 ;
4)      SQLHENV env1 ;
5)      SQLRETURN ret1, ret2, ret3, ret4 ;
6)      ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7)      if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8)      if (!ret2) ret3 = SQLConnect(con1, "dbs", SQLJTS, "js", SQL_NTS, "xyz", SQL_NTS) else exit ;
9)      if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10)     SQLPrepare(stmt1, "select UNOME, SALÁRIO from EMPREGADO where DNO = ?", SQL_NTS) ;
11)     prompt("Entre com o Numero do Departamento: ", dno) ;
12)     SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &fetchlen1) ;
13)     ret1 = SQLExecute(stmt1) ;
14)     if (!ret1) {
15)         SQLBindCol(stmt1, 1, SQL_CHAR, &unome, 15, &fetchlen1) ;
16)         SQLBindCol(stmt1, 2, SQL_FLOAT, &salario, 4, &fetchlen2) ;
17)         ret2 = SQLFetch(stmt1) ;
18)         while (!ret2) {
19)             printf(unome, salario) ;
20)             ret2 = SQLFetch(stmt1) ;
21)         }
22)     }
23) }
```

FIGURA 9.12 Segmento de programa CLI2, um segmento de programa C que usa a SQL/CLI para uma consulta cujo resultado apresenta uma coleção de tuplas.

9.5.2 JDBC: Chamadas de Função SQL em Programação JAVA

Voltaremos nossa atenção, agora, para como a SQL pode ser chamada na linguagem de programação orientada a objeto JAVA. As bibliotecas de funções para esse acesso são conhecidas como JDBC. A linguagem de programação JAVA foi

14 Uma técnica alternativa, conhecida como coluna não-associada (*unbound columns*), usa uma função diferente SQL/CLI, denominada SQLGetCol ou SQLGetData, que recupera as colunas do resultado da consulta sem que, anteriormente, seja feita uma associação entre elas; estas são aplicadas depois do comando SQLFetch no passo 17.

15 Se as variáveis de programa não-associadas (*unbound variables*) forem usadas, o SQLFetch devolverá a tupla em uma área temporária de programa. Cada SQLGetCol (ou SQLGetData) subsequente devolverá, na ordem, um valor de atributo.

16 Esta seção pressupõe familiaridade com os conceitos de orientação a objeto e com os conceitos básicos de JAVA. Se o leitor não possuir essa familiaridade, ele poderá postergar esta seção até terminar a leitura do Capítulo 20.

17 Como mencionamos anteriormente, o JDBC é uma marca registrada da Sun Microsystems, embora seja geralmente entendido como um acrônimo de Java Data Base Connectivity.

200 Capítulo 9 Mais SQL: Asserções (Assertions), Visões (Views) e Técnicas de Programação

projetada para ser independente da plataforma, ou seja, um programa deve rodar em qualquer tipo de sistema de computador que tenha um intérprete JAVA instalado. Por causa dessa portabilidade, muitos fabricantes de SGBDR fornecem drivers de JDBC para que programas JAVA possam acessar seus sistemas. Um *driver* JDBC é basicamente uma implementação das chamadas de função especificadas no JDBC API (Interface para a programação de aplicações) para o SGBDR de um fabricante em particular. Consequentemente, um programa JAVA, com chamadas de função JDBC, pode acessar qualquer SGBDR que tiver um driver JDBC disponível.

Como JAVA é orientado a objeto, suas bibliotecas de funções são implementadas como classes. Antes de poder processar as chamadas de função JDBC com JAVA, é necessário importar as bibliotecas de classes JDBC, que são chamadas `java.sql.*`. Elas podem ser carregadas e instaladas pela Web.

A JDBC foi projetada para permitir que um único programa JAVA pudesse conectar-se a vários bancos de dados diferentes. Estes, algumas vezes, são chamados fontes de dados, e acessados pelo programa JAVA. Essas fontes de dados podem ser armazenadas em SGBDRs de diferentes fabricantes, e podem residir em máquinas diferentes. Por conseguinte, as fontes de dados diferentes, acessadas dentro de um mesmo programa JAVA, podem exigir drivers JDBC de diferentes fabricantes. Para alcançar essa flexibilidade, é empregada uma classe especial JDBC, chamada classe gerente de *drivers*, que mantém o controle dos drivers instalados. Um *driver* deve ser *registrado* pelo gerente de drivers antes de ser usado. As operações (métodos) da classe de um gerente de drivers incluem `getDriver`, `registerDriver` e `deregisterDriver`. Elas podem ser usadas para adicionar ou remover os drivers dinamicamente. Outras funções marcam e encerram as conexões com as fontes de dados, como veremos adiante.

Para carregar um driver JDBC explicitamente, pode ser usada uma função JAVA genérica. Por exemplo, para carregar o driver JDBC no SGBD ORACLE, pode ser usado o seguinte comando:

```
Class.forName("oracle.jdbc.driver.OracleDriver")
```

Assim, o driver será registrado pelo gerente de drivers, que o tornará disponível para o programa. Também é possível carregar e registrar um(ns) driver(s), quando for necessário, na linha de comando que roda o programa; por exemplo, incluindo o seguinte comando:

```
-Djdbc.drivers = oracle.jdbc.driver
```

Os passos típicos em um programa de aplicação JAVA com acesso por chamadas de função JDBC a um banco de dados são os seguintes. Ilustraremos esses passos recorrendo ao exemplo JDBC1 da Figura 9.13, que lê o número do seguro social de um empregado e imprime seu último nome e salário.

1. A biblioteca de classes JDBC deve ser importada para o programa JAVA. Essas classes são chamadas `java.sql.*` e podem ser importadas usando a linha 1 da Figura 9.13. Qualquer biblioteca de classes adicional, necessária ao programa JAVA, também poderá ser importada.

2. Carregue o *driver* JDBC conforme discutido previamente (linhas 4 a 7). A exceção JAVA da linha 5 acontece se o *driver* não for devidamente carregado.

3. Crie variáveis apropriadas, conforme necessário ao programa JAVA (linhas 8 e 9).

4. Um objeto de conexão é criado utilizando-se a função `getConnection` da classe JDBC `DriverManager`. Nas linhas 12 e 13, o objeto de conexão é criado usando-se a chamada de função `getConnection` (uri string), em que uri string tem a forma

```
jdbc:oracle:<driverTipo>:<dbconta>/<senha>
```

Uma forma alternativa é

```
getConnection(url, dbconta, senha)
```

Diversas propriedades podem ser atribuídas (set) para um objeto de conexão, mas elas estão relacionadas, principalmente, às propriedades transacionais que discutiremos no Capítulo 17.

5. Um objeto de declaração é criado no programa. Em JDBC, há uma classe básica de declaração, `Statement`, com duas subclasses especializadas: `PreparedStatement` e `CallableStatement`. Esse exemplo ilustra como os objetos da `PreparedStatement` são criados e utilizados. O próximo exemplo (Figura 9.14) ilustra o outro tipo de objetos de declaração. Na linha 14, uma cadeia de caracteres com uma consulta de um único parâmetro — indicado pelo símbolo " V — é criada na variável `stmtl`. Na linha 15, um objeto `p`, do tipo `PreparedStatement`, é criado baseado na consulta `stmtl`, usando o objeto de conexão `conn`. De modo geral, o programador deve optar pelo objeto `PreparedStatement` se uma consulta for executada diversas vezes, uma vez que ela será escrita, verificada e compilada apenas uma vez, reduzindo, assim, os custos adicionais de execuções da consulta.

18 Estão disponíveis em diversos sites na Web — por exemplo, URL <http://industry.java.sun.com/products/jdbc/drivers>.

9.5 Programação em um Banco de Dados por meio de Funções (Functions Calls): SQL/CLI e JDBC 201

```
//Programa JDBC1:
0) import java.io.* ;
1) import java.sql.*;
2) class getEmpInfo {
3)     public static void main (String args []) throws SQLException,
IOException {
4)         try { Class.forName("oracle.jdbc.driver.OracleDriver");
5)         } catch (ClassNotFoundException x) {
6)             System.out.println ("Driver nao pode ser carregado") ;
7)         }
8)         String dbacct, senha, ssn, unome ;
9)         Double salário ;
10)        dbacct = readentry("Entre com conta do banco de dados:") ;
11)        passwd = readentry("Entre com a senha:") ;
12)        Connection conn = DriverManager.getConnection
13)            ("jdbc:oracle:oci8:" + dbacct + "/" + passwd) ;
14)        String stmt1 = "select UNOME, SALÁRIO from EMPREGADO where SSN = ?" ;
15)        PreparedStatement p = conn.prepareStatement(stmt1) ;
16)        ssn = readentry("Entre com o Numero do Seguro Social: ") ;
17)        p.clearParameters() ;
18)        p.setString(1, ssn) ;
19)        ResultSet r = p.executeQuery();
20)        while (r.next()){
21)            unome = r.getString(1);
22)            salário = r.getDouble(2) ;
23)            system.out.println(unome + salário) ;
24)        } }
25) }
```

FIGURA 9.13 Segmento de programa JDBC1, um segmento de programa JAVA com o JDBC.

6. O ponto de interrogação (?) da linha 14 representa um parâmetro de declaração, que é um valor a ser determinado em tempo de execução, normalmente ligado a uma variável do programa JAVA. Em geral, pode haver vários parâmetros, diferenciados pela ordem em que aparecem os pontos de interrogação na declaração (primeiro ? representa o parâmetro 1; segundo ?, o parâmetro 2, e assim por diante), como previamente discutido.

7. Antes de executar uma consulta com PreparedStatement, todos os parâmetros devem estar carregados em variáveis do programa. Dependendo do tipo dos parâmetros, funções como `setString`, `setInteger`, `setDouble`, e assim por diante são aplicadas ao objeto PreparedStatement para fixar seus parâmetros. Na Figura 9.13, o parâmetro (indicado por ?) do objeto p é ligado à variável `ssn` do programa JAVA na linha 18. Se houver n parâmetros na declaração SQL, devemos ter n `Set...functions`, cada uma com uma posição de parâmetro diferente (1,2, ..., n). Normalmente, essa forma é recomendada para tornar claros todos os parâmetros antes de fixar qualquer novo valor (linha 17).

8. Seguindo esses procedimentos, podemos executar a declaração SQL referida pelo objeto p usando a função `executeQuery` (linha 19). Há uma função genérica de execução (*execute*) em JDBC, mais duas funções especializadas: `executeUpdate` e `executeQuery`. A `executeUpdate` é usada em declarações SQL para a inserção, a exclusão ou a atualização, e devolve um valor inteiro que indica o número de tuplas que foram afetadas. A `executeQuery` é usada nas declarações SQL para recuperação de dados e retorna um objeto do tipo `ResultSet`, que discutiremos adiante.

9. Na linha 19, o resultado da consulta é devolvido em um objeto r do tipo `ResultSet`. Ele se assemelha a uma matriz bidimensional ou a uma tabela, em que as tuplas são as linhas e os atributos devolvidos, as colunas. Um objeto `ResultSet` é semelhante a um cursor de SQL embutido e a um iterador em SQLJ. Em nosso exemplo, quando a consulta é executada, r se refere à tupla anterior à primeira tupla do resultado da consulta. A função `r.next()` (linha 20) desloca a posição para a próxima tupla (linha) do objeto `ResultSet`, e devolve null se não houver mais nenhuma tupla. Isso é feito para controlar o laço. O programador pode se referir aos atributos da tupla corrente usando diversas *get... functions*, que dependem do tipo de cada atributo (por exemplo, `getString`, `getInteger`, `getDouble`, e assim por diante). O programador pode usar ou as posições dos atributos (1,2) ou os seus nomes reais ("UNOME", "SALÁRIO") com *get... functions*. Em nossos exemplos, usamos a anotação posicional nas linhas 21 e 22.

Em geral, o programador pode checar as exceções SQL depois de cada chamada função JDBC.

202 Capítulo 9 Mais SQL: Asserções (Assertions), Visões (Views) e Técnicas de Programação

Note que, ao contrário de algumas outras técnicas, o JDBC não faz distinções entre as consultas que devolvem uma única tupla e aquelas que retornam várias tuplas. Isso se justifica porque um resultado com uma única tupla é apenas um caso especial.

No exemplo JDBC1, a consulta SQL seleciona uma *única tupla*, assim o laço das linhas 20 a 24 é executado no máximo uma vez. O próximo exemplo, mostrado na Figura 9.14, ilustra a recuperação de várias tuplas. O segmento de programa JDBC2 lê (entrada) um número de departamento e então recupera os empregados que trabalham nesse departamento. Um laço então varre todos os registros dos empregados, um de cada vez, e imprime o último nome e salário do empregado. Esse exemplo também ilustra como podemos executar uma consulta diretamente, sem preparações prévias, como no exemplo anterior. Essa técnica é preferida para as consultas que são executadas uma só vez, pois a programação é mais simples. Na linha 17 da Figura 9.14, o programador cria um objeto Statement (em vez de PreparedStatement, como no exemplo anterior) sem associá-lo a uma cadeia de caracteres de uma consulta em particular. A cadeia da consulta *q* é passada ao objeto statement *s* quando é executado na linha 18.

Isso conclui nossa breve introdução ao JDBC. O leitor interessado deve consultar o site <http://java.sun.com/docs/books/tutorial/jdbc/>, que contém detalhes de JDBC avançado.

//Segmento de Programa JDBC2:

```

0)    import java.io.* ;
1)    import java.sql.*;
2)    class printDepartmentEmps {
3)        public static void main (String args []) throws SQLException,
IOException {
4)            try { Class.forName("oracle.jdbc.driver.OracleDriver");
5)                } catch (ClassNotFoundException x) {
6)                    System.out.println ("O Driver nao pode ser carregado") ;
7)                }
8)            String dbacct, senha, unome ;
9)            Double salário ;
10)           Integer dno ;
11)           dbacct = readentry("Entre com a conta do banco de dados:") ;
12)           passwd = readentry("Entre com a senha:") ;
13)           Connection conn = DriverManager.getConnection
14)               ("jdbc:oracle:oci8:" + "dbacct" + "/" + passwd) ;
15)           dno = readentry("Entre com o Numero do Departamento: ") ;
16)           String q = "select UNOME, SALÁRIO from EMPREGADO where DNO = " +
dno.toString() ;
17)           Statement s = conn.createStatement();
18)           ResultSet r = s.executeQuery(q) ;
19)           while (r.next()) {
20)               unome = r.getString(1);
21)               salary = r.getDouble(2) ;
22)               system.out.println(unome + salário) ;
23)           } }
24)       }

```

FIGURA 9.14 Segmento de programa JDBC2, um segmento de programa JAVA que usa o JDBC para uma consulta que tem, como resultado, uma coleção de tuplas.

9.6 PROCEDIMENTOS ARMAZENADOS EM BANCO DE DADOS (STORED PROCEDURES) E O SQL/PSM

Concluiremos este capítulo com dois tópicos adicionais relacionados à programação de banco de dados. Na Seção 9.6.1 discutiremos o conceito de procedimentos armazenados (*stored procedures*), que são módulos de programa armazenados pelo SGBD no servidor de banco de dados. A seguir, na Seção 9.6.2, abordaremos as extensões da SQL, especificadas no padrão, que contemplam os construtores para a programação de propósito geral em SQL. Essas extensões são conhecidas como SQL/PSM (*SQL/Persistent Stored Modules*) e podem ser usadas para escrever os procedimentos armazenados. O SQL/PSM também serve como um exemplo de linguagem de programação de banco de dados que estende o modelo e a linguagem de um banco de dados — denominada SQL — com alguns construtores de programação, como declarações condicionais e laços.

9.6 Procedimentos Armazenados em Banco de Dados (Stored Procedures) e o SQL/PSM 203

9.6.1 Procedimentos Armazenados em um Banco de Dados e Funções

Até agora, em nossa apresentação de técnicas de programação para um banco de dados, estava implícita a suposição de que o programa de aplicação de banco de dados estaria sendo executado em uma máquina-cliente, diferentemente da máquina em que o servidor — e principal parte do software SGBD — estaria rodando. Embora isso seja satisfatório para muitas aplicações, às vezes é útil criar módulos — programas de procedimentos de banco de dados ou funções — que são armazenados e executados pelo SGBD no servidor de banco de dados. Estes são historicamente conhecidos como **procedimentos armazenados** em banco de dados, embora possam ser funções ou procedimentos. O termo, usado no padrão SQL para os procedimentos armazenados, é **módulos armazenados de modo persistente**, dado que esses programas são armazenados pelo SGBD de modo persistente, de forma semelhante aos dados persistentes armazenados pelo SGBD. Os procedimentos armazenados são úteis nas seguintes circunstâncias:

- Se um programa de banco de dados é necessário para várias aplicações, pode ser armazenado no servidor e invocado por quaisquer dos programas de aplicação. Isso reduz a duplicação de esforços e melhora a modularidade do software.
- Executar um programa no servidor pode reduzir a transferência de dados e, conseqüentemente, em certas situações, os custos de comunicação entre os clientes e o servidor.

• Esses procedimentos podem aumentar o poder de modelagem proporcionado pelas visões, permitindo que tipos mais complexos derivados dos dados possam tornar-se disponíveis aos usuários dos bancos de dados. Além disso, podem ser usados para verificar as restrições mais complexas, que vão além do poder de especificação das asserções e dos gatilhos.

Em geral, muitos dos SGBDs comerciais permitem que os procedimentos armazenados e as funções possam ser escritos em uma linguagem de programação de propósito geral. Alternativamente, um procedimento armazenado pode ser criado a partir de comandos simples de SQL, como recuperações e atualizações. A forma geral de declarar procedimentos armazenados é a seguinte:

```
CREATE PROCEDURE <nome do procedimento ( <parâmetros> ) <declarações locais> <corpo do procedimento ;
```

Os parâmetros e as declarações locais são opcionais, e só são especificados se for preciso. Para declarar uma função, é necessário um tipo de retorno, assim a forma de declaração é:

```
CREATE FUNCTION <nome da função ( <parâmetros> ) RETURNS <tipo de retorno <declarações locais> <corpo da função>;
```

Se o procedimento (ou a função) é escrito em uma linguagem de programação de propósito geral, é normal especificar a linguagem, bem como um nome de arquivo em que o código do programa será armazenado. Por exemplo, o seguinte formato pode ser usado:

```
CREATE PROCEDURE <nome do procedimento ( <parâmetros> ) LANGUAGE <nome da linguagem de programação EXTERNAL NAME <nome do caminho do arquivo>;
```

Em geral, cada parâmetro deveria ser de um **tipo de parâmetro**, que é um dos tipos de dados SQL. Cada parâmetro também deveria ter um **modo de parâmetro**, que poderia ser IN, OUT ou INOUT. Estes correspondem a parâmetros cujos valores são: só entrada, saída (retorna) ou entrada e saída, respectivamente.

Como os procedimentos e as funções são armazenados de modo persistente pelo SGBD, deve ser possível chamá-los pelas diversas interfaces e técnicas de programação SQL. A declaração CALL, no padrão SQL, pode ser usada para invocar um procedimento armazenado — por meio de uma interface interativa, por uma SQL embutida ou por uma SQLJ. O formato da declaração é a seguinte:

```
CALL <nome do procedimento ou função ( <lista de argumentos> ) ;
```

Se essa declaração for chamada por JDBC, deveria ser designada a um objeto do tipo declaração CallableStatement (Seção 9.5.2).

9.6.2 SQL/PSM: Estendendo a SQL para a Especificação de Módulos Armazenados de Modo Persistente

O SQL/PSM é a parte do padrão SQL que especifica como escrever os módulos armazenados de modo persistente. Inclui as declarações para criar as funções e os procedimentos que descrevemos na seção anterior. Também inclui os construtores adicionais de programação para aumentar o poder da SQL para a escrita do código (ou do corpo) dos procedimentos armazenados e das funções. Nesta seção discutiremos os construtores SQL/PSM para as declarações condicionais (desvios) e para as declarações de laços (*loops*). Eles darão a noção dos construtores incorporados pelo SQL/PSM. Daremos, então, um exemplo para ilustrar como esses construtores podem ser usados.

A declaração para desvio condicional em SQL/PSM tem a seguinte forma :

```
IF <condição> THEN <lista de declarações>
ELSEIF <condição> THEN <lista de declarações>
ELSEIF <condição> THEN <lista de declarações>
ELSE <lista de declarações>
ENDIF;
```

Considere o exemplo da Figura 9.15, que ilustra como a estrutura de desvio condicional pode ser usada em uma função SQL/PSM. A função devolve um valor de cadeia de caracteres (linha 1) que descreve o tamanho de um departamento baseado no número de empregados. Há um parâmetro inteiro IN, deptnro, que fornece o número do departamento. Uma variável local NroDeEmps é declarada na linha 2. A consulta, das linhas 3 e 4, devolve o número de empregados do departamento, e o desvio condicional das linhas 5 a 8 devolverá, então, um dos valores {"ENORME", "GRANDE", "MÉDIO", "PEQUENO"}, baseado no número de empregados.

O SQL/PSM tem vários construtores para laços. Há as estruturas-padrão *while* (enquanto) e *repeat* (repetição), que têm as seguintes formas:

```
WHILE <condições> DO
<lista de declarações> END WHILE;
REPEAT
<lista de declarações> UNTIL <condição> END REPEAT ;
```

//Função PSM1:

```
0) CREATE FUNCTION DeptTamanho(IN deptnro INTEGER)
1) RETURNS VARCHAR [7]
2) DECLARE NroDeEmps INTEGER ;
3) SELECT COUNT(*) INTO NroDeEmps
4) FROM EMPREGADO WHERE DNO = deptnro ;
5) IF NroDeEmps > 100 THEN RETURN "ENORME"
6) ELSEIF NroDeEmps > 25 THEN RETURN "GRANDE"
7) ELSEIF NroDeEmps > 10 THEN RETURN "MEDIO"
8) ELSE RETURN "PEQUENO"
9) END IF ;
```

FIGURA 9.15 Declarando uma função em SQL/PSM.

Também há uma estrutura de laço baseada em cursor. A lista de declaração, em cada volta, é executada para cada tupla do resultado da consulta, uma por vez. Possui a seguinte forma:

19 Só daremos uma breve introdução ao SQL/PSM aqui. Há muitas outras características no padrão SQL/PSM.

9.7 Resumo 205

FOR <nome do laço> AS <nome do cursor> CURSOR FOR <consulta> DO

<lista de declarações> END FOR;

Os laços podem ter nomes e há uma declaração LEAVE <nome do laço> para quebrar um laço quando uma condição for satisfeita. A extensão SQL/PSM tem muitas outras características, mas elas estão fora do escopo de nossa apresentação.

9.7 RESUMO

Neste capítulo apresentamos as facilidades adicionais da linguagem de um banco de dados SQL. Em particular, demos uma visão geral das técnicas mais importantes para a programação de um banco de dados. Começamos na Seção 9.1, apresentando as facilidades para a especificação de restrições gerais, como as asserções. Logo após, discutimos o conceito de visão em SQL. Depois discutimos os vários enfoques para a programação de aplicações de um banco de dados nas seções 9.3 a 9.6.

Questões para Revisão

- 9.1. Como a SQL implementa as restrições de integridade gerais?
- 9.2. O que é uma visão em SQL e como é definida? Discuta os problemas que podem surgir quando se tenta atualizar uma visão. Como as visões são implementadas normalmente?
- 9.3. Relacione as três principais abordagens para a programação de um banco de dados. Quais são as vantagens e as desvantagens de cada uma?
- 9.4. O que é o problema de impedância de correspondência? Qual das três abordagens de programação minimiza esse problema?
- 9.5. Descreva o conceito de um cursor e como é usado em SQL embutida.
- 9.6. Para que é usada a SQLJ? Descreva os dois tipos de iteradores disponíveis em SQLJ.

Exercícios

- 9.7. Considere o banco de dados mostrado na Figura 1.2 cujo esquema é apresentado na Figura 2.1. Escreva um segmento de programa para ler o nome de um estudante e imprimir sua média de pontuação, presumindo que A=4, B=3, C=2 e D= 1 ponto. Use a SQL embutida em C (linguagem hospedeira).
- 9.8. Repita o Exercício 9.7 usando a SQLJ com JAVA como linguagem hospedeira.
- 9.9. Considere o esquema de um banco de dados relacional BIBLIOTECA da Figura 6.12. Escreva um segmento de programa que recupere a lista de livros com prazo vencido, e não pago, e imprima, para cada um, o título do livro e o nome do presta-tário. Use a SQL embutida em C.
- 9.10. Repita o Exercício 9.9, mas use a SQLJ com JAVA.
- 9.11. Repita os Exercícios 9.7 e 9.9 utilizando a SQL/CLI com C.
- 9.12. Repita os Exercícios 9.7 e 9.9 usando o JDBC com JAVA.
- 9.13. Repita o Exercício 9.7 escrevendo uma função em SQL/PSM.
- 9.14. Especifique as seguintes visões em SQL, para o esquema de um banco de dados EMPRESA, mostrado na Figura 5.5.
 - a. Uma visão que tenha o nome do departamento, o nome do gerente e o salário do gerente de todos os departamentos.
 - b. Uma visão que tenha o nome do empregado, o nome de seu supervisor e o salário de cada empregado que trabalhe no departamento de 'Pesquisa'.
 - c. Uma visão que tenha o nome do projeto, o nome do departamento responsável por seu controle, o número de empregados e as horas totais alocadas no projeto, por semana.
 - d. Uma visão que tenha o nome do projeto, o nome do departamento que o controla, o número de empregados e as horas totais trabalhadas, por semana, naqueles projetos *que tenham mais de um empregado alocado*.
- 9.15. Considere a seguinte visão, DEPT_RESUM0, definida para o banco de dados de EMPRESA da Figura 5.6:

```
CREATE VIEW      DEPT_RESUM0 (D, C, TOTAL_S, MEDIA_S)
AS SELECT      DNO, COUNT (*), SUM (SALÁRIO), AVG (SALÁRIO)
FROM            EMPREGADO
GROUP          BY
```

DNO;

206 Capítulo 9 Mais SQL: Asserções (Assertions), Visões (Views) e Técnicas de Programação

Estabeleça quais, dentre as seguintes consultas e atualizações, seriam permitidas como visão. Se uma consulta ou uma atualização for permitida, mostre a consulta ou a atualização correspondente, feita pelas relações básicas que mais se pareceriam com elas, e dê seus resultados quando aplicadas ao banco de dados da Figura 5.6.

- a. **SELECT***
FROM DEPT_RESUMO;
- b. **SELECT D, C**
FROM DEPT_RESUMO WHERE TOTAL_S > 100000;
- c. **SELECT D, MEDIA_S FROM DEPT_RESUMO**
WHERE C > (SELECT C FROM DEPT_RESUMO WHERE D=4);
- d. **UPDATE DEPT_RESUMO SET D=3**
WHERE D=4;
- e. **DELETE FROM DEPT_RESUMO WHERE C>4;**

Bibliografia Seleccionada

A questão das atualizações de visão é estabelecida por Dayal e Bernstein (1978), Keller (1982) e Langerak (1990), entre outros. As implementações de visão são discutidas em Blakeley *et al.* (1989). Negri *et al.* (1991) descreve a semântica formal *dt* consultas em SQL.

IP^{111*}

3

**TEORIA E METODOLOGIA DE
PROJETO DE UM BANCO DE DADOS**

10

Dependência Funcional e Normalização em um Banco de Dados Relacional

Dos capítulos 5 ao 9, apresentamos vários aspectos do modelo relacional e das linguagens a ele associadas. Cada *esquema de relação* consiste em um número de atributos, e o *esquema de banco de dados relacional* consiste em um determinado número de esquemas de relação. Adiante, pressupomos que, usando o bom senso do projetista do banco de dados, esses atributos serão agrupados para formar um esquema de relação ou será feito o mapeamento do esquema do projeto, de um modelo de dados conceitual como o ER ou o ER estendido (EER), ou algum outro modelo de dados conceitual. Esses modelos fazem com que o projetista identifique os tipos entidade e relacionamento e seus respectivos atributos, que levam a um agrupamento natural e lógico dos atributos em relações, quando os procedimentos para o mapeamento do Capítulo 7 forem seguidos. Entretanto, ainda precisaremos de alguma medida formal para saber por que um esquema de agrupamento de atributos em uma relação é melhor que outro. Na sequência de nossa discussão de projeto conceitual nos capítulos 3 e 4 e seu mapeamento para o modelo relacional no Capítulo 7, não desenvolvemos, além da intuição do projetista, nenhum critério de adequabilidade ou de 'excelência' para medir a qualidade do projeto. Neste capítulo discutiremos algumas das teorias desenvolvidas com o objetivo de avaliar a qualidade dos projetos de esquemas conceituais — isto é, medir formalmente por que um conjunto de agrupamentos de atributos em esquemas de relação é melhor que outro.

Existem dois níveis em que podemos discutir a 'excelência' dos esquemas de relação. O primeiro é o **nível lógico** (ou **conceitual**) — como os usuários interpretam os esquemas de relação e o significado de seus atributos. Ter bons esquemas de relação nesse nível possibilita que os usuários entendam claramente o significado dos dados nas relações e, assim, possam formular corretamente suas consultas. O segundo é o **nível de implementação** (ou **armazenamento**) — como as tuplas de uma relação básica são armazenadas e atualizadas. Esse nível se aplica somente aos esquemas de relações básicas — que são armazenadas fisicamente em arquivos —, embora, no nível lógico, estejamos interessados tanto nos esquemas de relações básicas quanto nas visões (relações virtuais). A teoria para projeto de um banco de dados relacional, desenvolvida neste capítulo, está mais voltada para as *relações básicas*, ainda que alguns dos critérios de adequabilidade também sejam apropriados às visões, como mostrado na Seção 10.1.

Diante desses problemas, o projeto de um banco de dados pode ser desenvolvido usando-se duas abordagens: *bottom-up* (ascendente) ou *top-down* (descendente). A **metodologia de projeto bottom-up** (também chamada *projeto por síntese*) considera os relacionamentos básicos *entre os atributos individuais* ponto de partida, e os utiliza para construir os esquemas de relações. Essa abordagem não é muito popular na prática porque sofre o problema de ter de coletar um grande número de relacionamentos binários entre os atributos como ponto de partida. Em contraste, a abordagem da **metodologia de projeto top-down** (também chamada *projeto por análise*) começa com um número de agrupamentos de atributos em relações

¹ Uma exceção, na qual essa abordagem é usada na prática, é o modelo chamado modelo relacional binário. Um exemplo é a metodologia N1AM (Verheijen e VanBekkum, 1982).

210 Capítulo 10 Dependência Funcional e Normalização em um Banco de Dados Relacional

agrupadas naturalmente, por exemplo, como um pedido, um formulário ou um relatório. As relações são, então, analisadas individual e coletivamente, indicando as decomposições futuras até que todas as propriedades sejam alcançadas. A teoria descrita neste capítulo é aplicada tanto para as abordagens de projeto *top-down* quanto para as abordagens *bottom-up*, mas é mais prática quando usada na abordagem *top-down*.

Começaremos este capítulo discutindo informalmente, na Seção 10.1, alguns critérios para avaliar se um esquema de relações é bom ou ruim. Então, na Seção 10.2, definiremos os conceitos de *dependência funcional*, uma restrição formal entre os atributos que é a principal ferramenta para a mensuração formal da adequabilidade do agrupamento dos atributos nos esquemas de relações. As propriedades da dependência funcional também serão estudadas e analisadas. Na Seção 10.3 mostraremos como as dependências funcionais podem ser usadas para agrupar os atributos em esquemas de relações que estão em *formas normais*. Um esquema de relação está em uma forma normal quando satisfizer certas propriedades desejáveis. O processo de *normalização* consiste na análise das relações para aumentar as formas normais estritas, levando a agrupamentos de atributos progressivamente melhores. As formas normais são especificadas em termos de dependências funcionais — que são identificadas pelo projetista do banco de dados — e de atributos-chave dos esquemas de relação. Na Seção 10.4 discutiremos as definições genéricas das formas normais, que podem ser diretamente aplicadas a um dado projeto e não exigem análise normalização passo a passo.

O Capítulo 11 continua o desenvolvimento da teoria relacionada ao projeto de bons esquemas relacionais. Embora no Capítulo 10 nos concentremos nas formas normais para um único esquema de relação, no Capítulo 11 discutiremos as medidas de conveniência para um conjunto de esquemas de relações como um todo, que formam juntos o esquema de um *banco de dados relacional*. Especificaremos duas dessas propriedades — a propriedade de junção não aditiva (sem perda) e a propriedade da preservação da dependência — e discutiremos os algoritmos para projetos *bottom-up* de um banco de dados relacional, que começam com um determinado conjunto de dependências funcionais e alcançam certas formas normais, e ainda mantêm as propriedades anteriormente mencionadas. Será também apresentado um algoritmo genérico que testa se uma decomposição possui ou não a propriedade da junção sem perda (Algoritmo 11.1). No Capítulo 11 poderemos definir os tipos adicionais de dependências e as formas normais avançadas que aprofundam a 'excelência' dos esquemas de relações.

Para os leitores interessados somente em uma introdução informal de normalização, as seções 10.2.3, 10.2.4 e 10.2.5 não precisam ser lidas. Se, em um curso, o Capítulo 11 não for tratado, recomendamos uma introdução rápida às propriedades de decomposição na Seção 11.1 e uma discussão da Propriedade LJ1, além do Capítulo 10.

10.1 ORIENTAÇÕES PARA PROJETOS INFORMAIS DE ESQUEMAS RELACIONAIS

Discutiremos, nesta seção, quatro *medidas informais* para mensurar a qualidade de um projeto de esquema de relação:

- Semântica dos atributos.
- Redução de valores redundantes nas tuplas.
- Redução de valores *nul* nas tuplas.
- Impedimento para a geração de valores ilegítimos nas tuplas.

Essas medidas nem sempre são independentes, como poderemos ver.

10.1.1 Semântica dos Atributos da Relação

Sempre que agrupamos os atributos para formar um esquema de relação, pressupomos que esses atributos possuam algum significado no mundo real e que haja uma interpretação própria associada a eles. No Capítulo 5 discutimos de que maneira cada relação pode ser interpretada como um conjunto de fatos ou declarações. Esse significado, ou semântica, especifica como interpretar os valores dos atributos armazenados em uma tupla da relação — em outras palavras, como os valores dos atributos de uma tupla se relacionam uns com os outros. Se o modelo conceitual for feito cuidadosamente, seguindo a sistemática do mapeamento para as relações, a maioria da semântica terá sido considerada e o modelo resultante deverá apresentar um significado claro.

Em geral, quanto mais fácil for explicar a semântica da relação, melhor será o modelo de esquema da relação. Para ilustrar, considere a Figura 10.1 - uma versão simplificada do esquema EMPRESA do banco de dados relacional da Figura 5.5, e a Figura 10.2 um exemplo de estado das relações preenchidas desse esquema. O significado do esquema da relação EMPREGADO é bastante simples: cada tupla representa um empregado, com valores para nome (ENOME), número do seguro social (SSN), data de nascimento (DATANASC), endereço (ENDERECO) e número do departamento em que o empregado trabalha (DNUMERO). O atributo DNUMERO é uma chave estrangeira que representa uma *relação implícita* entre EMPREGADO e DEPARTAMENTO. A semântica dos esquemas

10.1 Orientações para Projetos Informais de Esquemas Relacionais 211

DEPARTAMENTO e PROJETO também é direta: cada tupla de DEPARTAMENTO representa uma entidade departamento, e cada tupla PROJETO representa uma entidade projeto. O atributo DGERSSN de DEPARTAMENTO relaciona um departamento ao empregado que é seu gerente, enquanto DNUM de PROJETO relaciona um projeto a seu departamento controlador; ambos são atributos de chaves estrangeiras. A facilidade com que o significado dos atributos de uma relação pode ser explicado é uma *medida informal* de quão bem a relação foi projetada.

A semântica dos outros dois esquemas de relação, na Figura 10.1, é ligeiramente mais complexa. Cada tupla em DEPTLOCALIZACAO dá, a um número de departamento (DNUMERO), o local do departamento (DLOCALIZACAO). Cada tupla em TRABALHA_EM fornece, a um número de seguro social de empregado (SSN), O número de *um dos* projetos em que o empregado trabalha (PNUMERO), e o número de horas por semana que o empregado trabalha naquele projeto (HORAS). Porém, ambos os esquemas têm interpretações bem definidas e sem ambigüidades. O esquema DEPT_LOCALIZACOES representa um atributo multivalorado de DEPARTAMENTO, ao passo que TRABALHA_EM representa uma relação N:M entre EMPREGADO e PROJETO. Conseqüentemente, todos os esquemas de relação na Figura 10.1 podem ser considerados fáceis de explicar e, por conseguinte, bons do ponto de vista de possuírem uma semântica clara. Podemos, assim, formular as seguintes diretrizes informais para os projetos.

DIRETRIZ 1. Modelar um esquema de relação de modo que seja fácil explicar seu significado. Não combine os atributos de diferentes tipos entidades e relacionamentos dentro de uma única relação. Intuitivamente, se um esquema de relação corresponder a um tipo entidade ou a um tipo relacionamento, haverá uma justificativa direta para seu significado. Caso contrário, se a relação corresponder a uma mistura de entidades e relacionamentos, resultarão em ambigüidades semânticas e a relação não poderá ser explicada facilmente.

EMPREGADO			chave estrangeira (f.k.)	
ENOME	SSN	DATANASC	ENDEREÇO	DNUMERO

chave primária (p.k.)

DEPARTAMENTO		
DNOME	DNUMERO	DGERSSN

chave primária (p.k.)

DEPT_LOCALIZACOES	_____	chave estrangeira (f.k.)
DNUMERO	DLOCALIZACAO	

chave primária (p.k.)

PROJETO

chave estrangeira (f.k.)

PNUMERO	PNUMERO	PLOCALIZACAO	DNUM
---------	---------	--------------	------

chave primária (p.k.)

TRABALHA_EM chave estrangeira (f.k.)

SSN	PNUMERO	HORAS
-----	---------	-------

chave primária (p.k.)

FIGURA 10.1 Um esquema simplificado do banco de dados relacional EMPRESA.

Os esquemas das relações das figuras 10.3a e 10.3b também têm semântica clara. (Por hora, o leitor deverá ignorar as linhas abaixo das relações; elas serão usadas para ilustrar a anotação de dependência funcional, discutida na Seção 10.2.) Uma tupla do esquema da relação EMP_DEPT, na Figura 10.3a, representa um único empregado, mas inclui informações adicionais — isto é, o nome do departamento (DNOME) para o qual o empregado trabalha e o número do seguro social do gerente desse departamento (DGERSSN). Para a relação EMP_PROJ da Figura 10.3b, cada tupla relaciona um empregado a um projeto, mas também inclui o nome do empregado (ENOME), O nome do projeto (PNUMERO) e o local do projeto (PLOCALIZACAO). Embora não haja nada

212 Capítulo 10 Dependência Funcional e Normalização em um Banco de Dados Relacional

logicamente errado com essas duas relações, elas são consideradas esquemas pobres porque violam a Diretriz 1, misturando atributos de entidades distintas do mundo real; EMP_DEPT mistura os atributos de empregados e departamentos; e EMP_PROJ, atributos de empregados e projetos. Elas podem ser usadas como visões, porém causam problemas quando utilizadas como relações básicas, como discutiremos na seção seguinte.

EMPREGADO

ENOME	SNN	DATANASC	ENDEREÇO	DNUMERO

Smith,John B. Wong,FranklinT. Zelaya,Alicia J. Wallace,Jennifer S. Narayan,Remesh K. English,Joyce A. Jabbar,Ahmad V. Borg,James E.
 123456789 333445555 999887777 987654321 666884444 453453453 987987987 888665555
 1965-01-09 1955-12-08 1968-07-19 1941-06-20 1962-09-15 1972-07-31 1969-03-29 1937-11-10

DEPARTAMENTO

DNOME	DNUMERO	DGERSSN

Pesquisa
 Administração
 Diretoria
 333445555 987654321 888665555

TRABALHA EM

SNN	PNUMERO	HORAS

123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	null

FIGURA 10.2 Exemp Io de estado do e Squem

731 Fondren,Houston.TX 638 Voss,Houston.TX 3321 Castle,Spring.TX 291 Berry,Bellaire.TX 975 Fire Oak,Humble,TX 5631 Rice,Houston.TX 980
 Dallas,Houston,TX 450 Stone.Houston.TX
 5 5 4 4 5 5 4 1

DEPT.LOCALIZACOES

DNUMERO DLOCALIZACAO

Houston
 Stafford
 Bellaire
 Sugarland
 Houston

PROJETO

PNUMERO	PLOCALIZACAO	DNUM
ProdutoX		
ProdutoY		
ProdutoZ		
Automação		
Reorganização		
1		
2 3 10 20		
NovosBenefícios	30	
Bellaire		
Sugarland		
Houston		
Stafford		

10.1.2 Informações Redundantes em Tuplas e Anomalias de Atualizações

Uma meta para o modelo de um esquema é minimizar o espaço de armazenamento usado pelas relações básicas (e, conseqüentemente, dos arquivos correspondentes). Agrupar os atributos em esquemas de relações tem um efeito significativo em espaço de armazenamento. Por exemplo, compare o espaço usado pelas duas relações básicas EMPREGADO e DEPARTAMENTO, da Fig 10.2, com as relações básicas EMP_DEPT da Figura 10.4, que são o resultado da aplicação de uma operação de NATURAL JOIN entre EMPREGADO e DEPARTAMENTO. Em EMP_DEPT, os valores do atributo pertencente a um departamento em particular (DNUMERO, DNO DGERSSN) serão repetidos para *toda empregado que trabalhe naquele departamento*. Em contraste, a informação de cada departamento só aparece uma vez na relação DEPARTAMENTO da Figura 10.2. Só o número do departamento (DNUMERO) é repetido na relação EMPREGADO para cada empregado que trabalhe naquele departamento. Observações semelhantes aplicam-se à relação EMP_PROJ (Figura 10.4), que contém os atributos adicionais de EMPREGADO e PROJETO em relação a TRABALHA_EM.

10.1 Orientações para Projetos Informais de Esquemas Relacionais

213

(a)

EMP_DEPT

ENOME	SSN	DATANASC	ENDEREÇO	DNUMERO	DNOME	DGERSSN
\bar{r}	\bar{n}	\bar{r}	\bar{r}			

(b)

Dependência Funcional 1 (DF1) Dependência Funcional 2 (DF2) Dependência Funcional 3 (DF3)

EMP_PROJ

SSN	PNUMERO	HORAS	ENOME	PNOME	PLOCALIZACAO
F1)	\bar{n}				

FIGURA 10.3 Dois esquemas de relações que sofrem anomalias de atualização.

EMP_DEPT

redundância \bar{A}

ENOME	SSN	DATANASC	ENDEREÇO	DNUMERO	DNOME	DGERSSN
Smith,John B.	123456789	1965-01-09	731 Fondren,Houston,TX	5	Pesquisa	333445555
Wong,Franklin T.	333445555	1955-12-08	638 Voss,Houston,TX	5	Pesquisa	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle.Spring,TX	4	Administração	987654321
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4	Administração	987654321
Narayan,Ramesh K.	666884444	1962-09-15	975 FireOak,Humble,TX	5	Pesquisa	333445555
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5	Pesquisa	333445555
Jabbar,Ahmad V.	987987987	1969-03-29	980 Dallas,Houston,TX	4	Administração	987654321
Borg,James E.	888665555	1937-11-10	450 Stone,Houston,TX	1	Sede Administrativa	888665555

Redundância

redundância

PUD Dnni

A

X

SSN	PNUMERO	HORAS	ENOME	PNOME	PLOCALIZACAO
123456789	1	32.5	Smith,John B.	ProdutoX	Bellaire
123456789	2	7.5	Smith,John B.	ProdutoY	Sugarland
666884444	3	40.0	Narayan,Ramesh K.	ProdutoZ	Houston
453453453	1	20.0	English,Joyce A.	ProdutoX	Bellaire
453453453	2	20.0	English,Joyce A.	ProdutoY	Sugarland
333445555	2	10.0	Wong,Franklin T.	ProdutoY	Sugarland
333445555	3	10.0	Wong,Franklin T.	ProdutoZ	Houston
333445555	10	10.0	Wong,Franklin T.	Automação	Stafford
333445555	20	10.0	Wong,Franklin T.	Reorganização	Houston
999887777	30	30.0	Zelaya,Alicia J.	NovosBenefícios	Stafford
999887777	10	10.0	Zelaya,Alicia J.	Automação	Stafford
987987987	10	35.0	Jabbar,Ahmad V.	Automação	Stafford
987987987	30	5.0	Jabbar,Ahmad V.	Novos benefícios	Stafford
987654321	30	20.0	Wallace,Jennifer S.	Novos benefícios	Stafford
987654321	20	15.0	Wallace,Jennifer S.	Reorganização	Houston
888665555	20	null	Borg,James E.	Reorganização	Houston

FIGURA 10.4 Exemplo de estado para EMP_DEPT e EMP_PROJ resultantes da aplicação do NATURAL JOIN nas relações da Figura 10.2. Elas podem ser armazenadas como relações básicas por razões de desempenho.

Outro sério problema, quando se usam as relações da Figura 10.4 como relações básicas, é o de anomalias na atualização. Esses problemas podem ser classificados em anomalias de inserção, anomalias de exclusão e anomalias de atualização.

Anomalias de Inserção. As anomalias de inserção podem ser classificadas em dois tipos, ilustrados nos exemplos seguintes, tendo por base a relação EMP_DEPT:

- Para inserir uma nova tupla empregado em EMP_DEPT, teremos de incluir os valores dos atributos para o departamento que o empregado trabalha, ou *nulls* (se o empregado ainda não trabalhar em nenhum departamento). Por exemplo,

Essas anomalias foram identificadas por Codd (1972a) para justificar a necessidade da normalização de relações, como discutiremos na Seção 10.3.

214 Capítulo 10 Dependência Funcional e Normalização em um Banco de Dados Relacional

para inserir uma nova tupla de um empregado que trabalhe no departamento 5, precisaremos informar os valores dos atributos do departamento 5 corretamente, por isso eles devem ser *consistentes* com os demais valores do departamento 5 das outras tuplas em EMP_DEPT. NO modelo da Figura 10.2, não teremos de nos preocupar com esse problema de consistência porque entraremos somente com o número do departamento na tupla empregado; todos os outros valores de atributo do departamento 5 só serão registrados uma vez no banco de dados, com uma única tupla na relação DEPARTAMENTO.

- É difícil inserir um novo departamento que ainda não tenha nenhum empregado em EMP_DEPT. O único modo de fazer isso é colocar os valores nulos nos atributos do empregado. Isso causa problema porque SSN é chave primária de EMP_DEPT e supõe-se que cada tupla represente uma entidade empregado — não uma entidade departamento. Além disso, quando o primeiro empregado for designado àquele departamento, não precisaremos mais dessa tupla com os valores *null*. Esse problema não acontece no modelo da Figura 10.2, pois um departamento será inserido na relação DEPARTAMENTO mesmo que nenhum empregado trabalhe nele, e sempre que um empregado for designado para aquele departamento, uma tupla correspondente será inserida em EMPREGADO.

Anomalias de Exclusão. O problema de anomalias de exclusão está relacionado à segunda situação de anomalia de inserção discutida anteriormente. Ao apagarmos uma tupla de empregado em EMP_DEPT, pode ser que ela represente o último empregado que trabalhe naquele departamento, então, as informações relativas àquele departamento serão perdidas do banco de dados. Esse problema não acontece no banco de dados da Figura 10.2 porque as tuplas de DEPARTAMENTO serão armazenadas separadamente.

Anomalias de Atualização. Em EMP_DEPT, se mudarmos o valor de um dos atributos de um departamento em particular — digamos, o gerente do departamento 5 —, teremos de atualizar as tuplas de todos os empregados que trabalham naquele departamento; caso contrário, o banco de dados se tornará inconsistente. Se deixarmos de atualizar alguma tupla, um mesmo departamento poderá apresentar mais de um valor para o gerente em diferentes tuplas de empregados, o que poderia estar errado.

Com base nas três anomalias precedentes, podemos declarar a seguinte diretriz.

DIRETRIZ 2. Modelar esquemas de relações básicas de forma que nenhuma anomalia de inserção, exclusão ou alteração possa ocorrer nas relações. Se houver a possibilidade de ocorrer alguma anomalia, registre-a claramente e tenha certeza de que os programas que atualizam o banco de dados operarão corretamente.

A segunda diretriz é consistente, de certo modo, com as demais declarações da primeira diretriz. Podemos ver também a necessidade de uma abordagem mais formal para avaliar se um modelo cumpre essas diretrizes. As seções 10.2 a 10.4 abordam os conceitos formais necessários. É importante observar que essas diretrizes, às vezes, *precisam ser violadas* de modo a *melhorar o desempenho* de certas consultas. Por exemplo, se uma consulta importante recupera informações relativas ao departamento de um empregado, bem como os atributos do empregado, o esquema EMP_DEPT pode ser usado como relação básica. Porém, as anomalias de EMP_DEPT devem ser notadas e consideradas (por exemplo, o uso de gatilhos ou procedimentos armazenados que façam atualizações automáticas) sempre que a relação básica for atualizada, para que não terminemos gerando inconsistências. Em geral, é aconselhável usar as relações básicas livres de anomalias e especificar as visões que estabeleçam junções para unir os atributos freqüentemente referidos em consultas importantes. Isso reduz o número de condições de JOIN especificadas na consulta, tornando mais simples a redação correta da consulta e, em muitos casos, melhorando o desempenho.

10.1.3 Valores Nulls (Nulos) em Tuplas

Em alguns projetos de esquemas, podemos agrupar muitos atributos em uma relação 'gorda'. Se muitos dos atributos não se aplicarem a todas as tuplas da relação, acabaremos com muitos *nulls* nessas tuplas. Isso pode causar desperdício de espaço no armazenamento, além de gerar problemas de entendimento do significado dos atributos e da especificação de operações de JOIN no nível lógico. Outro problema com os *nulls* é como tratá-los em operações agregadas como SUM ou COUNT. Além disso, os *nulls* podem ter diversas interpretações, tais como:

- 3 Esse não é um problema tão sério quanto os demais, pois todas as tuplas podem ser atualizadas por uma única consulta SQL.
- 4 O desempenho de uma consulta específica em uma visão, que é a junção de várias relações básicas, depende de como o SGBD implementa essa visão. Muitos SGBDRs materializam uma visão usada com freqüência, de modo que não precisem executar a junção freqüentemente. Resta ao SGBD a responsabilidade de atualizar as visões materializadas (imediate ou periodicamente) sempre que as relações básicas forem atualizadas.
- 5 Isso porque as junções internas e externas produzem resultados diferentes quando envolvem os *nulls*. Assim, os usuários devem ficar atentos aos diferentes tipos de junção. Embora possam ser razoáveis para usuários sofisticados, podem se tornar difíceis para os demais.

10.1 Orientações para Projetos Informais de Esquemas Relacionais 215

- Que o atributo *não se aplica* à tupla.
- O valor do atributo para a tupla é *desconhecido*.
- O valor é *conhecido, mas ausente*; ou seja, ainda não foi registrado.

Uma vez que todos os *nulls* têm a mesma representação, compromete-se os diversos significados que eles podem assumir. Assim, podemos estabelecer outra diretriz.

DIRETRIZ 3. Até onde for possível, evite colocar os atributos em uma relação básica cujos valores frequentemente possam ser nulos. Se os *nulls* forem inevitáveis, tenha certeza de que eles se aplicam somente em casos excepcionais e não na maioria das tuplas da relação.

O uso eficiente do espaço e a redução de junções são os dois critérios que determinam a inclusão de colunas que possam ter os *nulls* em uma relação ou optar por uma relação separada para essas colunas (com as chaves apropriadas). Por exemplo, se só 10% dos empregados tiverem escritórios particulares, há pouca justificativa para incluir um atributo ESCRITORIO_NRO na relação EMPREGADO; pode ser criada uma relação EMP_ESCRITORIOS (ESSN, ESCRITORIO_NRO) que contenha apenas as tuplas dos empregados que possuem escritórios particulares.

10.1.4 Geração de Tuplas Ilegítimas

Considere os dois esquemas de relação EMP_LOCS e EMP_PROJ da Figura 10.5a que podem ser usados em vez da relação única EMP_PROJ da Figura 10.3b. Uma tupla em EMP_LOCS significa que o empregado cujo nome é ENOME trabalha em *algum projeto* cujo local é PLOCALIZACAO. Uma tupla de EMP_PROJ 1 significa que o empregado cujo número do seguro social é SSN trabalha HORAS por semana no projeto cujo nome, número e local são PNAME, PNUMERO e PLOCALIZACAO. A Figura 10.5b apresenta os estados das relações EMP_LOCS e EMP_PROJ1, que correspondem à relação EMP_PROJ da Figura 10.4, e que são obtidos aplicando-se as operações PROJECT (ir — projecção) apropriadas em EMP_PROJ (por hora, ignore as linhas pontilhadas da Figura 10.5b).

Suponha que, em vez de EMP_PROJ, usemos EMP_PROJ1 e EMP_LOCS como relações básicas. Isso produz um projeto de esquema particularmente ruim porque não poderemos recuperar, a partir de EMP_PROJ1 e EMP_LOCS, as informações que originalmente havia em EMP_PROJ. Se tentarmos uma operação de NATURAL JOIN em EMP_PROJ1 e EMP_LOCS, o resultado produzirá muito mais tuplas que o conjunto original de tuplas de EMP_PROJ. Na Figura 10.6 é mostrado o resultado da aplicação de uma junção nas tuplas *acima* das linhas pontilhadas da Figura 10.5b (apenas para reduzir o tamanho da relação resultante). As tuplas adicionais, que não estavam em EMP_PROJ, são chamadas tuplas ilegítimas porque representam as informações ilegítimas ou *erradas*, que não são válidas. As tuplas ilegítimas são marcadas com asteriscos (*) na Figura 10.6.

A decomposição de EMP_PROJ em EMP_LOCS e EMP_PROJ1 é indesejável, pois, quando as rearranjarmos usando NATURAL JOIN, não obteremos as informações originais corretas. Isso porque, nesse caso, PLOCALIZACAO é o atributo que relaciona EMP_LOCS e EMP_PROJ1, e PLOCALIZACAO não é nem uma chave primária nem uma chave estrangeira em EMP_LOCS ou EMP_PROJ1. Podemos agora, informalmente, estabelecer outra diretriz de projeto.

DIRETRIZ 4. Projete os esquemas de relações de forma que possam ser unidos (*join*) com igualdade de condições sobre os atributos que sejam chaves primárias ou chaves estrangeiras, de modo a garantir que nenhuma tupla ilegítima seja gerada. Evite as relações que contenham o relacionamento entre os atributos que não sejam combinações (chave estrangeira, chave primária), porque as junções sobre esses atributos podem produzir tuplas ilegítimas.

É óbvio que, essa diretriz informal precisa ser especificada mais formalmente. No Capítulo 11 discutiremos uma condição formal, chamada propriedade de junção não aditiva (ou sem perda), que garante que determinadas junções não produzam tuplas ilegítimas.

10.1.5 Resumo e Discussão das Diretrizes de Projeto

Nas seções 10.1.1 a 10.1.4 discutimos, informalmente, as situações que conduzem a esquemas de relações problemáticos, e propusemos diretrizes informais para um bom projeto relacional. Os problemas que apontamos podem ser descobertos sem ferramentas adicionais de análise, conforme veremos a seguir:

- Anomalias que causem trabalho redundante nas inserções e nas alterações de uma relação, e que possam causar perda acidental de informações em exclusões na relação.
- Desperdício de espaço de armazenamento por causa de *nulls* e dificuldade de executar as operações de agregação e junção em virtude de valores *nulls*.
- Geração de dados inválidos e ilegítimos em junções de relações básicas relacionadas indevidamente.

(a)

EMP_LOCS

ENOME	PLOCALIZACAO

chave primária (p.k.)

EMP_PROJ1

SSN	PNUMERO	HORAS	PNOME	PLOCALIZACAO

chave primária (p.k.)

(b)

EMP_LOCS

ENOME	PLOCALIZACAO
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wongj Franklin T.	Stafford
Zelaya, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

EMP_PROJ1

SSN	PNUMERO	HORAS	PNOME	PLOCALIZACAO
123456789	1	32.5	Produto X	Bellaire
123456789	2	7.5	Produto Y	Sugarland
666884444	3	40.0	Produto Z	Houston
453453453	1	20.0	Produto X	Bellaire
453453453	2	20.0	Produto Y	Sugarland
333445555	2	10.0	Produto Y	Sugarland
333445555	3	10.0	Produto Z	Houston
333445555	10	10.0	Automação	Stafford
333445555	20	10.0	Reorganização	Houston
999887777	30	30.0	Novos benefícios	Stafford
999887777	10	10.0	Automação	Stafford
987987987	10	35.0	Automação	Stafford
987987987	30	5.0	NovosBenefícios	Stafford
987654321	30	20.0	NovosBenefícios	Stafford
987654321	20	15.0	Reorganização	Houston
888665555	20	null	Reorganização	Houston

FIGURA 10.5 Projeto particularmente pobre para a relação EMP_PROJ da Figura 10.3b. (a) Os dois esquemas de relações EMP_LOCS e EMP_PROJ1. (b) O resultado da projeção de EMP_PROJ, da Figura 10.4, para as relações EMP_LOCS e EMP_PROJ1.

No restante deste capítulo, apresentaremos a teoria e os conceitos formais que podem ser usados para definir mais precisamente a 'excelência' e a 'imperfeição' dos esquemas *individuais* de uma relação em particular. Primeiro, discutiremos a dependência funcional como uma ferramenta para análise. Então, especificaremos as três formas normais e a forma normal de Boyce-Codd (BCNF) para os esquemas de relação. No Capítulo 11 definiremos as formas normais adicionais que são baseadas em outros tipos de dependências de dados chamadas dependências multivaloradas e dependências de junção.

10.2 Dependências Funcionais

217

SSN	PNUMERO	HORAS	PNOME	PLOCALIZACAO	ENOME
123456789	1	32.5	Produto X	Bellaire	Smith.John B.
123456789	1	32.5	Produto Y	Bellaire	English.Joyce A.
123456789	2	7.5	Produto Y	Sugarland	Smith.John B.
123456789	2	7.5	Produto Y	Sugarland	English.Joyce A.
123456789	2	7.5	Produto Y	Sugarland	Wong.FranklinT.
666884444	3	40.0	Produto Z	Houston	Narayan.Ramesh
666884444	3	40.0	Produto Z	Houston	Wong.FranklinT.
453453453	1	20.0	Produto X	Bellaire	Smith.John B.
453453453	1	20.0	Produto X	Bellaire	English.Joyce A.
453453453	2	20.0	Produto Y	Sugarland	Smith.John B.
453453453	2	20.0	Produto Y	Sugarland	English.Joyce A.
453453453	2	20.0	Produto Y	Sugarland	Wong.FranklinT.
333445555	2	10.0	Produto Y	Sugarland	Smith.John B.
333445555	2	10.0	Produto Y	Sugarland	English.Joyce A.
333445555	2	10.0	Produto Y	Sugarland	Wong.FranklinT.
333445555	3	10.0	Produto Z	Houston	Narayan.Ramesh
333445555	3	10.0	Produto Z	Houston	Wong.FranklinT.
333445555	10	10.0	Automação	Stafford	Wong.FranklinT.
333445555	20	10.0	Reorganização	Houston	Narayan.Ramesh
333445555	20	10.0	Reorganização	Houston	Wong.FranklinT.

FIGURA 10.6 Resultado da aplicação de NATURAL JOIN nas tuplas acima da linha pontilhada de EMP_PROJ1 e EMP_LOCS da Figura 10.5. As tuplas ilegítimas geradas estão marcadas com asteriscos.

10.2 DEPENDÊNCIAS FUNCIONAIS

O conceito mais importante da teoria de projetos de esquemas relacionais é o da dependência funcional. Nesta seção definiremos formalmente esse conceito e, na Seção 10.3, veremos como ele pode ser usado para especificar as formas normais para os esquemas relacionais.

10.2.1 Definição de Dependência Funcional

Uma dependência funcional é uma restrição entre dois conjuntos de atributos do banco de dados. Suponha que nosso esquema de banco de dados relacional tenha n atributos A_1, A_2, \dots, A_n ; vamos pensar em um banco de dados como um único esquema de relação **universal** $R = \{A_1, A_2, \dots, A_n\}$. Não podemos dizer que armazenaremos todo o banco de dados em uma única tabela universal; usaremos esse conceito apenas para o desenvolvimento da teoria formal de dependências de dados.

Definição. Uma **dependência funcional**, denotada por $X \rightarrow Y$ entre dois conjuntos de atributos X e Y , que são subconjuntos de R , especificam uma *restrição* nas possíveis tuplas que formem um estado da relação r de R . A restrição é que, para quaisquer duas tuplas t_1 e t_2 em r que tenham $t_1[X] = t_2[X]$, elas também têm de ter $t_1[Y] = t_2[Y]$.

Isso significa que os valores do componente de Y de uma tupla em r dependem dos, ou são *determinados por*, valores do componente X ; alternativamente, os valores do componente X da tupla *determinam* exclusivamente (ou **funcionalmente**) os valores do componente de Y . Também dizemos que há dependência funcional de X para Y ou que Y é **funcionalmente dependente** de X . A abreviação para a dependência funcional é DF ou d.f. O conjunto de atributos X é chamado o **lado à esquerda da DF**, e o de Y é dito o **lado à direita**.

Assim, X determina funcionalmente Y em um esquema de relação R se, e somente se, duas tuplas de $r(R)$ corresponderem em seus X -valores e necessariamente corresponderem em seus Y -valores. Observe o seguinte:

- Uma restrição nos estados de R , que implica não haver mais de uma tupla com um dado valor X em qualquer instância da relação $r(R)$ — isto é, X é uma chave **candidata** de R —, implica $X \rightarrow Y$ para qualquer subconjunto de atributos

6 Esse conceito de uma relação universal será importante quando discutirmos os algoritmos para os projetos de um banco de dados relacional no Capítulo 11.

7 Essa suposição sugere que todo atributo do banco de dados deveria ter um *nome distinto*. No Capítulo 5 prefixamos os nomes dos atributos com o nome da relação para alcançar a singularidade sempre que os atributos em relações diferentes tivessem o mesmo nome.

218 Capítulo 10 Dependência Funcional e Normalização em um Banco de Dados Relacional

Y de R [porque a restrição de chave implica que duas tuplas em qualquer estado válido $r(R)$ não terão valores iguais em **X**].

Se $X \rightarrow Y$ em R, isso não implica necessariamente que $Y \rightarrow X$ em R.

Uma dependência funcional é uma propriedade da **semântica** ou do **significado dos atributos**. Os projetistas de um banco de dados usarão sua compreensão da semântica dos atributos de R — isto é, como eles se relacionam uns com os outros — para especificar as dependências funcionais que devem sujeitar *todos* os estados da relação (extensões) de r em R. Sempre que a semântica de dois conjuntos de atributos em R indicar que uma dependência funcional deveria ser assegurada, especificaremos a dependência como uma restrição. As extensões de relações $r(R)$ que satisfizerem a restrição de dependência funcional são chamadas **estados legais da relação** (ou **extensões legais**) de R. Portanto, o principal uso das dependências funcionais é descrever um esquema de relação R por meio da especificação de restrições de seus atributos, que devem ser asseguradas o tempo *todo*. Certas DFs podem ser especificadas sem recorrer a uma relação específica, mas apenas pelas propriedades de seus atributos. Por exemplo, {ESTADO, CARTEIRA_HABILITACAO_NRO} \rightarrow SSN deveria ser garantida para qualquer adulto nos Estados Unidos. Também é possível que certas dependências funcionais possam deixar de existir no mundo real se houver mudanças no relacionamento. Por exemplo, a DF entre CEP_COD \rightarrow AREA_COD era usada para promover um relacionamento entre os códigos postais e os números de telefone nos Estados Unidos, mas com a proliferação dos códigos de área de telefone, isso já não é mais verdadeiro.

Considere o esquema da relação EMP_PROJ da Figura 10.3b; pela semântica dos atributos, sabemos que as seguintes dependências funcionais deveriam ser asseguradas:

- a. SSN \rightarrow ENOME
- b. PNUMERO \rightarrow {PNOME, PLOCALIZACAO}
- c. {SSN, PNUMERO} \rightarrow HORAS

Essas dependências funcionais especificam que: a) o valor de um número do seguro social de um empregado (SSN) determina, exclusivamente, o nome do empregado (ENOME); b) o valor de um número de projeto (PNUMERO) determina, exclusivamente, o nome do projeto (PNOME) e seu local (PLOCALIZACAO); e c) uma combinação entre os valores de SSN e PNUMERO determina exclusivamente, o número de horas por semana que o empregado trabalha no projeto (HORAS). Alternativamente, dizemos que ENOME é funcionalmente determinado por SSN (ou funcionalmente dependente de), ou 'com um valor de SSN determinado, sabemos o valor de ENOME', e assim por diante.

Uma dependência funcional é uma *propriedade do esquema da relação R*, não de um estado particular válido da relação de R. Logo, uma DF *não pode* ser deduzida automaticamente de uma determinada extensão da relação r , mas deve ser definida explicitamente por alguém que conheça a semântica dos atributos de R. Por exemplo, a Figura 10.7 mostra um estado particular do esquema da relação ENSINA. Embora, à primeira vista, possamos pensar que TEXTO \rightarrow CURSO, não poderemos confirmá-lo: a menos que saibamos ser verdade para todos os *possíveis estados válidos* de ENSINA. Entretanto, é suficiente demonstrar *apenas um contra-exemplo* para contestar uma dependência funcional. Por exemplo, como 'Smith' ensina tanto 'Estruturas de dados quanto Administração de dados', podemos concluir que PROFESSOR *não* determina funcionalmente o CURSO.

A Figura 10.3 introduz uma **notação diagramática** para as DFs: cada DF é exibida como uma linha horizontal. Os atributos à esquerda da DF estão conectados por linhas verticais à linha que representa a DF, enquanto os atributos à direita são conectados por setas que apontam para os atributos, como mostrado nas figuras 10.3a e 10.3b.

ENSINA

PROFESSOR CURSO TEXTO

Smith	Estruturas de dados	Bartram
Smith	Administração de dados	Al-Nour
Hall	Compiladores	Hoffman
Brown	Estruturas de dados	Augenthaler

FIGURA 10.7 Um estado da relação ENSINA com uma *possível* dependência funcional TEXTO \rightarrow CURSO. Embora PROFESSOR \rightarrow CURSO não seja confirmado.

10.2.2 Regras de Inferência para as Dependências Funcionais

Denotamos por F o conjunto de dependências funcionais que são especificadas no esquema da relação R. Em geral, o projetista do esquema especifica as dependências funcionais que são *semanticamente evidentes*, porém, normalmente, diversas outras

10.2 Dependências Funcionais 219

dependências funcionais existem em *todas* as instâncias válidas da relação que satisfazem as dependências em F. Essas outras dependências podem ser *deduzidas* ou *inferidas* das DFs de F.

Na vida real, é impossível especificar todas as possíveis dependências funcionais para uma determinada situação. Por exemplo, se cada departamento tem um gerente, de forma que DEPT_NO determine exclusivamente GERENTE_SSN (DEPT_NO \rightarrow GER_SSN), e um Gerente possui um número de telefone próprio GER_FONE (GER_SSN \rightarrow GER_FONE), então essas duas dependências induzem DEPT_NO \rightarrow GER_FONE. Trata-se de uma DF inferida e que *não* precisa ser declarada explicitamente, como as outras duas DFs o foram. Portanto, formalmente, é útil definir um conceito chamado *clausura*, que inclui todas as possíveis dependências que possam ser inferidas de um dado conjunto F.

Definição. Formalmente, o conjunto de todas as dependências de F, bem como todas as dependências que podem ser inferidas para F, é chamado **clausura** de F, que é denotada por F^+ .

Por exemplo, suponha que especifiquemos o seguinte conjunto F de dependências funcionais evidentes para o esquema de relação da Figura 10.3a:

$F = \{SSN \rightarrow \{ENOME, DATANASC, ENDEREÇO, DNUMERO\}, DNUMERO \rightarrow \{DNOME, DGERSSN\}\}$

Algumas das dependências funcionais adicionais que podemos inferir em F são as seguintes:

$SSN \rightarrow \{DNOME, DGERSSN\}$

$SSN \rightarrow SSN DNUMERO \rightarrow DNOME$

Uma **DF $X \rightarrow Y$ é inferida** de um conjunto de dependências F especificadas em R se $X \rightarrow Y$ ocorre para cada estado válido r da relação de R, ou seja, sempre que r satisfizer todas as dependências de F, $X \rightarrow Y$ também é assegurada para r. A clausura F^+ de F é o conjunto de todas as dependências funcionais que podem ser inferidas para F. Para determinar um modo sistemático de dedução de dependências, temos de descobrir um conjunto de **regras de inferência** que possa ser usado para deduzir novas dependências de um determinado conjunto de dependências. Consideremos algumas dessas regras de inferências a seguir. Usamos a notação $F \vdash X \rightarrow Y$ para indicar que as dependências funcionais $X \rightarrow Y$ foram inferidas para o conjunto de dependências funcionais F.

Na discussão seguinte, usaremos uma anotação abreviada para tratar as dependências funcionais. Concatenamos as variáveis de atributo e tiramos as vírgulas por conveniência. Consequentemente, a DF $\{X, Y\} \rightarrow Z$ será abreviada para $XY \rightarrow Z$, e a DF $\{X, Y, Z\} \rightarrow \{U, V\}$ será abreviada para $XYZ \rightarrow UV$. As seis regras seguintes, IR1 a IR6, são aquelas de inferência bastante difundida para as dependências funcionais:

IR1 (regra reflexiva⁸): Se $X \subseteq Y$, então $X \rightarrow Y$.

IR2 (regra aumentativa⁹): $\{X \rightarrow Y\} \vdash NXZ \rightarrow YZ$.

IR3 (regra transitiva): $\{X \rightarrow Y, Y \rightarrow Z\} \vdash NX \rightarrow Z$.

IR4 (regra de decomposição ou projetiva): $\{X \rightarrow YZ\} \vdash NX \rightarrow Y$.

IR5 (regra de união ou aditiva): $\{X \rightarrow Y, X \rightarrow Z\} \vdash NX \rightarrow YZ$.

IR6 (regra pseudotransitiva): $\{X \rightarrow Y, WY \rightarrow Z\} \vdash WX \rightarrow Z$.

A regra reflexiva (IR1) estabelece que um conjunto de atributos sempre se determina, ou qualquer um de seus subconjuntos, o que é óbvio. Como IR1 gera dependências que sempre são verdadeiras, essas dependências são chamadas *triviais*. Formalmente, uma dependência funcional $X \rightarrow Y$ é **trivial** se $X \subseteq Y$; caso contrário, é **não trivial**. A regra aumentativa (IR2) diz que a soma do mesmo conjunto de atributos em ambos os lados, esquerdo e direito, de uma dependência resulta em outra dependência válida. De acordo com IR3, dependências funcionais são transitivas. A regra de decomposição (IR4) diz que podemos remover atributos do lado direito de uma dependência; aplicando essa regra repetidamente, podemos decompor a DF $X \rightarrow \{A_1, A_2, \dots, A_n\}$ em um conjunto de dependências $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$. A regra de união (IR5) nos permite fazer o oposto: podemos combinar um conjunto de dependências $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$ em uma única DF $X \rightarrow \{A_1, A_2, \dots, A_n\}$.

8 A regra reflexiva também pode ser declarada como $X \rightarrow X$, ou seja, qualquer conjunto de atributos determina funcionalmente a si mesmo.

9 A regra aumentativa também pode ser declarada como $\{X \rightarrow Y\} \vdash NXZ \rightarrow Y$, ou seja, aumentando-se os atributos do lado esquerdo de uma DF, produz-se outra DF válida.

Uma nota de advertência ao uso dessas regras. Embora $X \rightarrow A$ e $X \rightarrow B$ implique $X \rightarrow AB$ pela regra de união declarada anteriormente, $X \rightarrow A$ e $Y \rightarrow B$ *não implica* que $XY \rightarrow AB$. Também, $XY \rightarrow A$ necessariamente *não* implica $X \rightarrow A$ ou $Y \rightarrow A$.

Cada uma das regras precedentes pode ser provada a partir da definição de dependência funcional, por meio de prova direta ou **por** contradição. A prova por meio de contradição pressupõe que a regra não seja garantida e mostra que ela não é possível. Provaremos agora que as primeiras três regras IR1 a IR3 são válidas. A segunda prova é feita por meio de contradição.

PROVA DE IR1

Suponha que $X \rightarrow Y$ e que existam duas tuplas t_1 e t_2 em alguma instância da relação r de R , tal que $t_1[X] = t_2[X]$. Então $t_1[Y] = t_2[Y]$ porque $X \rightarrow Y$; conseqüentemente, $X \rightarrow Y$ é assegurada em r .

PROVA DE IR2 (PELA CONTRADIÇÃO)

Pressuponha que $X \rightarrow Y$ seja assegurada para uma instância da relação r de R , mas que $XZ \rightarrow YZ$ não o seja. Então devem existir duas tuplas t_1 e t_2 em r tal que: 1) $t_1[X] = t_2[X]$; 2) $t_1[Y] = t_2[Y]$; 3) $t_1[XZ] \neq t_2[XZ]$; e 4) $t_1[YZ] \neq t_2[YZ]$. Isso não é possível porque de (1) e (3) deduzimos (5) $t_1[Z] = t_2[Z]$, e de (2) e (5) deduzimos (6) $t_1[YZ] = t_2[YZ]$, contradizendo (4).

PROVA DE IR3

Presuma que (1) $X \rightarrow Y$ e (2) $Y \rightarrow Z$ sejam ambas asseguradas para uma relação r . Então para quaisquer duas tuplas t_1 e t_2 em r tal que $t_1[X] = t_2[X]$, temos de ter (3) $t_1[Y] = t_2[Y]$ da hipótese (1); conseqüentemente, também temos de ter (4) $t_1[Z] = t_2[Z]$ de (3) e da suposição (2); por conseguinte, $X \rightarrow Z$ tem de ser assegurada para r .

Usando argumentos semelhantes, podemos provar as regras IR4 a IR6 e quaisquer regras adicionais de inferência válidas. Porém, um modo mais simples de provar que uma regra de inferência para as dependências funcionais é válida é prová-la usando as regras de inferência que já foram provadas. Por exemplo, podemos provar IR4 a IR6 usando IR1 a IR3, como segue.

PROVA DE IR4 (USANDO IR1 A IR3)

1. $X \rightarrow YZ$ (dado).
2. $YZ \rightarrow Y$ (usando IR1 e sabendo que $YZ \wedge Y$).
3. $X \rightarrow Y$ (usando IR3 em 1 e 2).

PROVA DE IR5 (USANDO IR1 A IR3)

1. $X \rightarrow Y$ (dado).
2. $X \rightarrow Z$ (dado).
3. $X \rightarrow XY$ (usando IR2 em 1 acrescido de X ; note que $X \wedge X = X$).
4. $XY \rightarrow YZ$ (usando IR2 em 2, acrescido de Y).
5. $X \rightarrow YZ$ (usando IR3 em 3 e 4).

PROVA DE IR6 (USANDO IR1 A IR3)

1. $X \rightarrow Y$ (dado).
2. $WY \rightarrow Z$ (dado).
3. $WX \rightarrow WY$ (usando IR2 em 1 acrescido de W).
4. $WX \rightarrow Z$ (usando IR3 em 3 e 2).

Foi demonstrado por Armstrong (1974) que as regras de inferência IR1 a IR3 são sólidas e completas. Por **sólida** queremos dizer que, dado um conjunto de dependências funcionais F especificado para um esquema de relação R , toda dependência que pudermos deduzir para F usando IR1 a IR3 será assegurada para qualquer relação estado r de R que *satisfizer a dependências* de F . Por **completa** queremos dizer que, se usarmos IR1 a IR3 sucessivamente para deduzir as dependências, até que mais nenhuma dependência possa ser deduzida, resultará no conjunto completo de *todas as dependências possíveis* que podem ser inferidas para F . Em outras palavras, o conjunto de dependências F , que chamamos **clausura** de F , pode ser determinado

10.2 Dependências Funcionais 221

utilizando-se apenas as regras de inferência IR1 a IR3. AS regras de inferência IR1 a IR3 são conhecidas como **regras de inferência de Armstrong**.

Em geral os projetistas de banco de dados especificam, primeiramente, o conjunto de dependências funcionais F que podem ser facilmente determinadas pela semântica dos atributos de R ; então IR1, IR2 e IR3 são usadas para deduzir dependências funcionais adicionais, que também serão garantidas para R . Um modo sistemático de especificar essas dependências funcionais adicionais é determinar primeiro os conjuntos de atributos X que aparecem do lado esquerdo de alguma dependência funcional F , e depois estabelecer o conjunto de *todos os atributos* que sejam dependentes de X . Assim, para cada um desses conjuntos de atributos X , determinamos o conjunto X^+ de atributos, que é funcionalmente determinado por X com base em F . X^+ é chamado **clausura de X** em F . O Algoritmo 10.1 pode ser usado para calcular X^+ .

Algoritmo 10.1:

Determinando X^+ , a Clausura de X em F ,

$X^+ := X$; repetir

$\text{old}X^+ := X^+$;

para cada dependência funcional $Y \rightarrow Z$ em F faça

se X^+DY então $X^+ := X^+UZ$; até ($X^+ = \text{old}X^+$);

O Algoritmo 10.1 começa fixando X para todos os atributos de X . Por IR1 sabemos que todos esses atributos são funcionalmente dependentes de X . Por meio das regras de inferência IR3 e IR4, acrescentamos atributos a X usando cada dependência funcional de F . Mantemos o procedimento para todas as dependências de F (o laço *repeat*) até que mais nenhum atributo possa ser acrescentado a X *após um ciclo completo* (do laço *for*) das dependências de F . Por exemplo, considere o esquema da relação EMP_PROJ da Figura 10.3b; da semântica dos atributos, especificamos o seguinte conjunto F de dependências funcionais que deveriam ser asseguradas para EMP_PROJ:

$F = \{SSN \rightarrow ENOME\}$

$PNUMERO \rightarrow \{PNOME, PLOCALIZACAO\}, \{SSN, PNUMERO\} \rightarrow HORAS\}$

Usando o Algoritmo 10.1 calculamos os seguintes conjuntos de clausura para F :

$\{SSN\}^+ = \{SSN, ENOME\}$

$\{PNUMERO\}^+ = \{PNUMERO, PNOME, PLOCALIZACAO\}$

$\{SSN, PNUMERO\}^+ = \{SSN, PNUMERO, ENOME, PNOME, PLOCALIZACAO, HORAS\}$

Intuitivamente, o conjunto de atributos do lado direito de cada linha representa todos os atributos que são funcionalmente dependentes do conjunto de atributos do lado esquerdo, com base em um dado conjunto F .

10.2.3 Equivalência de Conjuntos de Dependências Funcionais

Nesta seção discutiremos a equivalência de dois conjuntos de dependências funcionais. Primeiro, daremos algumas definições preliminares.

Definição. Diz-se que um conjunto de dependências funcionais F **cobre** outro conjunto de dependências funcionais E se toda DF em E também está em F , isto é, se toda dependência em E puder ser inferida em F ; alternativamente, podemos dizer que E é **coberto por F** .

Definição. Dois conjuntos de dependências funcionais E e F são **equivalentes** se $E^+ = F^+$. Conseqüentemente, a equivalência significa que toda DF em E pode ser inferida de F , e toda DF em F pode ser inferida de E , isto é, E é equivalente a F se ambas as condições forem asseguradas: E cobre F e F cobre E .

10 São, de fato, conhecidas como axiomas de Armstrong. No senso matemático estrito, os *axiomas* (fatos dados) seriam as dependências funcionais de F , uma vez que presumimos que elas são corretas, ao passo que IR1 a IR3 são as *regras de inferência* para deduzir novas dependências funcionais (fatos novos).

Podemos verificar se F cobre E calculando X com respeito a F para cada DF $X \rightarrow Y$ em E , e então verificar se esse X inclui os atributos em Y . Se for esse o caso para *toda* DF em E , então F cobre E . Determinamos se E e F são equivalentes checando se E cobre F e se F cobre E .

10.2.4 Conjuntos Mínimos de Dependências Funcionais

Informalmente, a **cobertura mínima** de um conjunto de dependências funcionais E é um conjunto de dependências funcionais F que satisfaça a propriedade que diz que toda dependência E pertence à clausura F^+ de F . Essa propriedade será perdida se qualquer dependência do conjunto F for removida; F não deve ter nenhuma redundância, e as dependências em E estão na forma-padrão. Para satisfazer essas propriedades, podemos definir formalmente que um conjunto de dependências funcionais F será **mínimo** se satisfizer as seguintes condições:

1. Toda dependência F tem um único atributo em seu lado direito.
2. Não podemos substituir nenhuma dependência $X \rightarrow A$ em F por uma dependência $Y \rightarrow A$, em que Y é um determinado subconjunto de X , e ainda ter um conjunto de dependências que seja equivalente a F .
3. Não podemos remover nenhuma dependência de F e ainda ter um conjunto de dependências que seja equivalente a F .

Podemos imaginar o conjunto mínimo de dependências como um conjunto de dependências numa *forma-padrão*, ou numa *forma canônica*, e sem *redundâncias*. A condição 1 apenas representa todas as dependências de uma forma canônica com um único atributo no seu lado direito. As condições 2 e 3 asseguram que não há nenhuma redundância nas dependências, seja por não possuírem os atributos redundantes do lado esquerdo da dependência (Condição 2), seja por terem dependências que possam ser inferidas das demais DFs de F (Condição 3). Uma **cobertura mínima** de um conjunto de dependências funcionais E é um conjunto mínimo de dependências F , que é equivalente a E . Pode haver várias coberturas mínimas para um conjunto de dependências funcionais. Sempre acharemos *pelo menos uma* cobertura mínima F para qualquer conjunto de dependências E usando o Algoritmo 10.2.

Se, pela definição anterior, vários conjuntos de DFs qualificarem-se como coberturas mínimas de E , é habitual usar os critérios adicionais para 'minimalidade'. Por exemplo, podemos escolher o conjunto mínimo com o *menor número de dependências* ou o *mais curto* (o comprimento total de um conjunto de dependências é calculado concatenando-se as dependências e tratando-as como uma cadeia de caracteres longa).

Algoritmo 10.2: Achando uma Cobertura Mínima F para um Conjunto de Dependências Funcionais E

1. Set $F := E$.
2. Substitua cada dependência funcional $X \rightarrow \{A_1, A_2, \dots, A_J\}$ em F pelas n dependências funcionais $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n$.
3. For cada dependência funcional $X \rightarrow A$ em F for cada atributo B que seja elemento de X
if $((F - (X \rightarrow A)) \cup ((X - B) \rightarrow A))$ for equivalente a F , then substitua $X \rightarrow A$ por $(X - \{B\}) \rightarrow A$ em F .
4. For cada dependência funcional restante $X \rightarrow A$ em F if $(F - (X \rightarrow A))$ for equivalente a F , then remova $X \rightarrow A$ de F .

No Capítulo 11 veremos como as relações podem ser sintetizadas, a partir de um dado conjunto de dependências E achando primeiro a cobertura mínima F para E .

10.3 FORMAS NORMAIS BASEADAS EM CHAVES PRIMÁRIAS

Tendo estudado as dependências funcionais e algumas de suas propriedades, estamos prontos para usá-las na especificação de alguns aspectos da semântica dos esquemas da relação. Presumimos que um conjunto de dependências funcionais seja

11 Esta é a forma-padrão para simplificar as condições e os algoritmos que asseguram não haver nenhuma redundância em F . Usando a regra da inferência IR4 podemos converter uma única dependência, com diversos atributos do lado direito, em um conjunto de dependências com um só atributo do lado direito.

* Foram mantidos, no pseudocódigo, os termos em inglês, comuns às linguagens de programação. (N. de T.)

10.3 Formas Normais Baseadas em Chaves Primárias 223

específico para cada relação, e que cada relação tenha uma chave primária designada; essas informações, combinadas com os testes (condições) para formas normais, direcionam o *processo de normalização* dos projetos de esquemas relacionais. A maioria dos projetos práticos relacionais considera uma das seguintes abordagens:

- Primeiro executa-se um projeto de esquema conceitual usando-se um modelo conceitual como ER ou EER, e então se mapeia o projeto conceitual em um conjunto de relações.
- Projetam-se as relações com base no conhecimento externo derivado de uma implementação existente de arquivos, formulários ou relatórios.

Seguindo qualquer uma dessas abordagens, é útil avaliar, então, a adequabilidade das relações, decompô-las conforme necessário, para alcançar formas normais mais altas, usando a teoria de normalização apresentada neste capítulo e no próximo. Focalizaremos, nesta seção, as três primeiras formas normais para os esquemas de relação e suas concepções, e discutiremos como elas foram desenvolvidas historicamente. Definições mais genéricas dessas formas normais, que, em vez de considerar somente a chave primária, como vimos até agora, levam em conta todas as chaves candidatas da relação, serão postergadas para a Seção 10.4.

Começaremos discutindo as formas normais e a motivação informal para seu desenvolvimento; também revisaremos algumas definições do Capítulo 5 que serão necessárias aqui. Discutiremos então a primeira forma normal (1FN) na Seção 10.3.4, e serão apresentadas as definições da segunda forma normal (2FN) e da terceira forma normal (3FN), que são baseadas nas chaves primárias, respectivamente nas seções 10.3.5 e 10.3.6.

10.3.1 Normalização de Relações

O processo de normalização, como foi inicialmente proposto por Codd (1972a), sujeita um esquema de relação a uma série de testes para 'certificar-se' de que ele satisfaça uma certa **forma normal**. O processo, que segue o estilo *top-down*, avalia cada relação sob os critérios de cada forma normal e as decompõe, se necessário, podendo ser considerado um *projeto relacional por análise*. Inicialmente, Codd propôs três formas normais que ele chamou de primeira, segunda e terceira forma normal. Uma definição mais forte da 3FN — chamada forma normal Boyce-Codd (FNBC ou BCNF) — foi depois proposta por Boyce e Codd. Todas essas formas normais são baseadas nas dependências funcionais entre os atributos de uma relação. Depois, uma quarta forma normal (4FN) e uma quinta forma normal (5FN) foram propostas, baseadas nos conceitos de dependências multivaloradas e de junção, respectivamente; esses conceitos são discutidos no próximo capítulo. No começo do Capítulo 11, discutiremos também como relações na 3FN podem ser sintetizadas a partir de um dado conjunto de DFs. Esse enfoque é chamado *projeto relacional por síntese*.

A **normalização de dados** pode ser vista como o processo de análise de determinados esquemas de relações com base em suas DFs e chaves primárias para alcançar as propriedades desejáveis: de (1) minimização de redundância e (2) minimização de anomalias de inserção, exclusão e atualização, discutidas na Seção 10.1.2. Os esquemas de relações insatisfatórios, que não alcançam certas condições — os **testes de forma normal** —, são decompostos em esquemas de relações menores que passam nos testes e, conseqüentemente, possuem as propriedades desejadas. Assim, o procedimento de normalização proporciona aos projetistas de banco de dados o seguinte:

- Uma estrutura formal para a análise de esquemas de relação, com base em suas chaves e nas dependências funcionais entre seus atributos.
- Uma série de testes de formas normais deve ser feita, em cada um dos esquemas de relação, de forma que o banco de dados relacional possa ser **normalizado** no grau desejado.

A **forma normal** de uma relação refere-se à condição da mais alta forma normal alcançada e, conseqüentemente, indica o grau no qual foi normalizada. As formas normais, quando consideradas *isoladamente* de outros fatores, não garantem um bom projeto de banco de dados. Em geral, não é suficiente checar separadamente se cada esquema de relação do banco de dados é, digamos, BCNF ou 3FN. Pelo contrário, o processo de normalização por decomposição também deve confirmar a existência de propriedades adicionais que o esquema relacional, como um todo, deveria possuir. Eles devem ter duas propriedades:

- As **propriedades da junção sem perda** ou **junção não aditiva**, que garante que o problema de geração de tuplas ilegítimas, discutido na Seção 10.1.4, não ocorra nos esquemas de relação criados após a decomposição.
- A **propriedade da preservação da dependência**, que garante que cada dependência funcional será representada em alguma relação individual resultante da decomposição.

A propriedade da junção não aditiva é extremamente crítica e deve ser alcançada a qualquer preço; já a propriedade da preservação da dependência, embora desejável, é às vezes sacrificada, como discutiremos na Seção 11.1.2. Adiaremos a apresentação dos conceitos e das técnicas formais, que garantem as duas propriedades anteriores, para o Capítulo 11.

10.3.2 Uso Prático das Formas Normais

A maioria dos projetos práticos lida com os projetos de bancos de dados existentes, projetos em modelos legados ou com arquivos existentes. A normalização, realizada na prática, garante que os projetos resultantes sejam de alta qualidade e alcancem propriedades desejáveis, definidas previamente. Embora existam várias formas normais de alto grau, como as 4FN e 5FN, que discutiremos no Capítulo 11, a utilidade prática dessas formas normais se torna questionável quando as restrições, nas quais são baseadas, são difíceis de serem entendidas ou detectadas pelos projetistas e pelos usuários de um banco de dados, os quais deveriam estabelecê-las. Assim, o projeto praticado hoje comercialmente dá particular atenção somente à normalização até a 3FN, 4FN ou BCNF. Outro ponto a notar é que os projetistas de um banco de dados *não precisam* normalizar até a forma normal mais alta possível. As relações podem permanecer em um estado de normalização mais baixo, como 2FN, por razões de desempenho, como discutidas no final da Seção 10.1.2. O processo de armazenamento de junções das relações de formas normais altas em relações básicas — as quais estão em formas normais mais baixas — é conhecido como desnormalização.

10.3.3 Definições de Chaves e Atributos Participantes das Chaves

Antes de continuarmos, vejamos novamente as definições de chave de um esquema de relação do Capítulo 5.

Definição. Uma **superchave** de um esquema de relação $R = \{A_1, A_2, \dots, A_n\}$ é um conjunto de atributos $S \subseteq R$ que contenha a propriedade na qual não haverá duas tuplas t_1 e t_2 , em qualquer estado válido da relação r de R , cuja $t_1[S] = t_2[S]$. Uma **chave** K é uma superchave com a propriedade adicional de que a remoção de qualquer atributo de K fará com que K não seja mais uma superchave.

A diferença entre uma chave e uma superchave é que uma chave tem de ser *mínima*, ou seja, se tivermos uma chave $K = \{A_1, A_2, \dots, A_j\}$ de R , então $K - \{A_i\}$ não será chave de R para qualquer A_i , $1 < i < k$. Na Figura 10.1, $\{SSN\}$ é uma chave para EMPREGADO, considerando que $\{SSN\}$, $\{SSN, ENOME\}$, $\{SSN, ENOME, DATANASC\}$, e qualquer conjunto de atributos que incluam SSN serão todos superchaves.

Se um esquema de relação tiver mais de uma chave, cada uma delas é chamada **chave candidata**. Uma das chaves candidatas é *arbitrariamente* designada para ser a **chave primária**, e as outras são conhecidas como chaves secundárias. Cada esquema de relação deve ter uma chave primária. Na Figura 10.1, $\{SSN\}$ é a única chave candidata de EMPREGADO e, portanto também é a chave primária.

Definição. Um atributo de um esquema de relação R é chamado **atributo primário** de R se for membro de alguma *chave candidata* de R . Um atributo é dito **não primário** se não for um atributo primário — isto é, se não for membro de alguma chave candidata.

Na Figura 10.1, ambos o SSN e o $PNUMERO$ são atributos primários de TRABALHA_EM, considerando que outros atributos de TRABALHA_EM são não primários.

Apresentaremos agora as três primeiras formas normais: 1FN, 2FN e 3FN. Elas foram propostas por Codd (1972a) como uma sequência para alcançar o estado desejável de relações na 3FN, passando, se necessário, pelos estados intermediários de 1FN e 2FN. Como veremos, a 2FN e a 3FN abordam problemas diferentes. Porém, por razões históricas, é habitual seguir essa sequência; conseqüentemente, presumiremos que uma relação na 3FN *já satisfaz* a 2FN.

10.3.4 Primeira Forma Normal

A **primeira forma normal (1FN)** é agora considerada parte da definição formal de uma relação no modelo relacional básico (*flat*); historicamente, foi definida como impedimento para a criação de atributos multivalorados, atributos compostos e combinações entre eles. Estabelece-se que o domínio de um atributo só deva incluir os *valores atômicos* (simples, indivisíveis), e que o *valor* de qualquer atributo em uma tupla deve ter um único *valor* no domínio daquele atributo. Conseqüentemente, a

12 Essa condição não é considerada no *modelo relacional aninhado* e em *sistemas objeto-relacionais* (SGBDORs), pois ambos permitem *relações não normalizadas* (Capítulo 22).

10.3 Formas Normais Baseadas em Chaves Primárias 225

1FN desaprova, como valor de atributo de *uma única tupla*, um conjunto de valores, uma tupla de valores ou uma combinação entre ambos. Em outras palavras, 1FN impede as 'relações dentro de relações' ou as 'relações como valores de atributo dentro de tuplas'.

Os únicos valores de atributos permitidos pelas 1FN são valores únicos, **atômicos** (ou **indivisíveis**).

Considere o esquema da relação DEPARTAMENTO mostrado na Figura 10.1, cuja chave primária é DNUMERO, e suponha que acrescentemos o atributo DLOCALIZACOES, como mostrado na Figura 10.8a. Pressupomos que cada departamento possa ter *várias* localizações. O esquema de DEPARTAMENTO, e um exemplo de estado da relação, são mostrados na Figura 10.8. Como podemos ver, ele não está na 1FN porque DLOCALIZACOES não é um atributo atômico, como ilustrado pela primeira tupla da Figura 10.8b. Há duas maneiras de olharmos o atributo DLOCALIZACOES:

(a)

DEPARTAMENTO

(b)

(c)

DNOME	DNUMERO	DGERSSN	DLOCALIZACOES
-------	---------	---------	---------------

t

—!

•

DEPARTAMENTO

DNOME	DNUMERO	DGERSSN	DLOCALIZACOES
-------	---------	---------	---------------

Pesquisa	5	Administração	333445555	{Bellaire, Sugarland, Houston}	{Stafford}
4 Diretoria	1		987654321	{Houston}	

DEPARTAMENTO

888665555

DNOME	DNUMERO	DGERSSN	DLOCALIZACAO

Pesquisa

Pesquisa

Pesquisa

Administração

Diretoria

333445555 333445555 333445555 987654321 888665555

Bellaire

Sugarland

Houston

Stafford

Houston

FIGURA 10.8 Normalização na 1FN. (a) Esquema de relação que não está na 1FN. (b) Exemplo de estado da relação DEPARTAMENTO, (c) Versão na 1FN de alguma relação com redundância.

- O domínio de DLOCALIZACOES contém os valores atômicos, mas algumas tuplas podem ter um conjunto desses valores. Nesse caso, DLOCALIZACOES *não* é funcionalmente dependente da chave primária DNUMERO.
- O domínio de DLOCALIZACOES contém um conjunto de valores e conseqüentemente não é atômico. Nesse caso, DNUMERO - > DLOCALIZACOES, porque cada conjunto é considerado um único membro do domínio do atributo.

De qualquer modo, a relação DEPARTAMENTO da Figura 10.8 não está na 1FN; na realidade, nem mesmo pode ser qualificada como uma relação, de acordo com a nossa definição de relação na Seção 5.1. Há três técnicas básicas para alcançar a primeira forma normal para uma relação:

1. Remover o atributo DLOCALIZACOES que viola a 1FN e colocá-lo em uma relação separada DEPT_LOCALIZACOES ao lado da chave primária DNUMERO de DEPARTAMENTO. A chave primária dessa relação será a combinação {DNUMERO, DLOCALIZACAO}, como mostrado na Figura 10.2. Existirá uma tupla distinta para *cada uma das localizações* de departamento em DEPT_LOCALIZACOES. Assim, a relação fora da 1FN será decomposta em duas relações na 1FN.

2. Ampliar a chave de forma a separar as tuplas da relação original DEPARTAMENTO, criando uma para cada localização de DEPARTAMENTO, como mostrado na Figura 10.8c. Nesse caso, a chave primária se torna a combinação {DNUMERO, DLOCALIZACAO}. Essa solução tem a desvantagem de introduzir *redundância* na relação.

3. Se *um número máximo de valores* puder ser estabelecido para o atributo — por exemplo, se é sabido que há *no máximo três locais* para cada departamento —, substituir o atributo DLOCALIZACOES por três atributos atômicos: DLOCALIZACAO 1, DLOCALIZACAO2 e DLOCALIZACAO3. Essa solução tem a desvantagem de introduzir muitos *valores nulls*, caso a maioria dos departamentos tenha menos de três locais. Introduzirá, posteriormente, uma semântica inválida quando for feita a ordenação dos valores de localização,

13 Nesse caso, podemos considerar que o domínio de DLOCALIZACOES é o *power set* dos conjuntos de localizações únicas, ou seja, o domínio é composto por todos os possíveis subconjuntos do conjunto de localizações únicas.

226 Capítulo 10 Dependência Funcional e Normalização em um Banco de Dados Relacional

que não foi pretendida originalmente. A consulta a esses atributos torna-se mais difícil; por exemplo, considere, nesse projeto, a consulta "Liste os departamentos que possuam 'Bellaire' como uma de suas localizações".

Das três soluções anteriores, geralmente a primeira é considerada a melhor porque não causa redundância e é completamente genérica, uma vez que não impõe nenhum limite para o número máximo de valores. Na realidade, se escolhermos a segunda solução, ela será decomposta nos próximos passos da normalização, chegando à primeira solução.

A primeira forma normal também desaprova os atributos multivalorados que sejam compostos. Estes são chamados **relações aninhadas**, pois há uma relação *dentro de cada* tupla. A Figura 10.9 mostra como a relação EMP_PROJ poderia aparecer se o aninhamento fosse permitido. Cada tupla representaria uma entidade empregado, e *dentro de cada tupla*, a relação PROJS (PNUMERO, HORAS), representaria os projetos do empregado e as horas por semana em que ele trabalha em cada projeto. O esquema dessa relação EMP_PROJ pode ser representado como segue:

EMP_PROJ (SSN, ENOME, {PROJS (PNUMERO, HORAS)})

(a)

EMP PROJ

SSN

ENOME

PROJS

PNUMERO HORAS

(b)

EMP PROJ

SSN	ENOME	PNUMERO	HORAS
123456789	Smith.John B.	1 2	32.5 7.5
666884444	Narayan.Ramesh K.	3	40.0
453453453	English.Joyce A.	1 2	20.0 20.0
333445555	Wong.FranklinT.	2 3 10 20	10.0 10.0 10.0 10.0
999887777	Zelaya,Alicia J.	30 10	30.0 10.0
987987987	Jabbar.Ahmad V.	10 30	35.0 5.0
987654321	Wallacejennifer S.	30 20	20.0 15.0

(c)

EMP_PROJ1

SSN	ENOME

EMP_PROJ2

SSN	PNUMERO	HORAS

FIGURA 10.9 Normalização de relações aninhadas na 1FN. (a) Esquema da relação EMP_PROJ com 'relações aninhadas' nos atributos PROJS. (b) Exemplo de extensão da relação EMP_PROJ mostrando as relações aninhadas dentro de cada tupla. (c) Decomposição de EMP_PROJ em relações EMP_PROJ1 e EMP_PROJ2 por meio da propagação da chave primária.

O conjunto de chaves { } indica que o atributo PROJS é multivalorado, e relacionamos entre parênteses () os atributos componentes de PROJS. É interessante observar que as tendências recentes para o suporte a objetos complexos (Capítulo 20) e dados XML (Capítulo 26) que usam o modelo relacional trabalham para permitir e formalizar as relações aninhadas dentro de sistemas de um banco de dados relacional, anteriormente desaprovadas por meio da 1FN.

Observe que SSN é a chave primária da relação EMP_PROJ nas figuras 10.9a e 10.9 b, enquanto PNUMERO é a **chave parcial** da relação aninhada, ou seja, cada tupla dentro da relação aninhada precisa ter valores únicos para PNUMERO. Para normalizá-la na 1FN, removemos os atributos da relação aninhada, formando uma nova relação e *propagando a chave primária*; a chave primária

10.3 Formas Normais Baseadas em Chaves Primárias 227

da nova relação será a combinação da chave parcial com a chave primária da relação original. A decomposição e a propagação da chave primária resultam nos esquemas EMP_PROJ1 e EMP_PROJ2, mostrados na Figura 10.9c.

Esse procedimento pode ser aplicado recursivamente em uma relação com os níveis de aninhamento múltiplos de modo a **decompor** a relação em um conjunto de relações na 1FN. Isso é útil para converter um esquema de relação não normalizado em muitos níveis aninhados de relações na 1FN. A existência de mais de um atributo multivalorado em uma relação deve ser controlada cuidadosamente. Como exemplo, considere a seguinte relação fora da 1FN:

PESSOA (NUM_SS, {NUM_CAR_LIC}, {NUM_TELEFONE})

Essa relação representa uma pessoa que pode ter diversos carros e telefones. Se uma estratégia como a da segunda opção descrita anteriormente fosse seguida, resultaria em uma relação com todos os seus atributos como chaves:

PESSOA_NA_1FN (NUM_SS, NUM_CAR_LIC, NUM_TELEFONE)

Para evitar introduzir qualquer relacionamento estranho entre NUM_CAR_LIC e NUM_TELEFONE, todas as combinações possíveis de valores serão representadas para qualquer NUM_SS, originando redundâncias. Isso induz os problemas tratados pelas dependências multivaloradas e 4FN, que discutiremos no Capítulo 11. A maneira correta de lidar com os dois atributos multivalorados em PESSOA é decompor a relação em duas relações, usando a estratégia 1, discutida previamente:

P1(NUM_SS, NUM_CAR_LIC) e P2 (NUM_SS, NUM_TELEFONE).

10.3.5 Segunda Forma Normal

A **segunda forma normal (2FN)** é baseada no conceito de *dependência funcional total*. Uma dependência funcional $X \rightarrow Y$ será uma **dependência funcional total** se a remoção de qualquer atributo A de X implicar que a dependência não mais será assegurada, isto é, para qualquer atributo $A \in X$, $(X - \{A\})$ não determina funcionalmente Y. Uma dependência funcional $X \rightarrow Y$ é uma **dependência parcial** se um atributo A $\notin X$ puder ser removido de X e a dependência mesmo assim continuar existindo, ou seja, para algum $A \in X$, $(X - \{A\}) \rightarrow Y$. Na Figura 10.3b, {SSN, PNUMERO} \rightarrow HORAS é uma dependência total (não são asseguradas nem SSN \rightarrow HORAS nem PNUMERO \rightarrow HORAS). Porém, a dependência {SSN, PNUMERO} \rightarrow ENOME é parcial porque SSN \rightarrow ENOME é assegurada.

Definição. Um esquema de relação R está na 2FN se todo atributo não primário A em R tem *dependência funcional total* da chave primária de R.

O teste para 2FN envolve verificar se os atributos do lado esquerdo das dependências funcionais fazem parte da chave primária. Se a chave primária contiver um único atributo, a necessidade do teste não se aplica. A relação EMP_PROJ, na Figura 10.3b, está na 1FN, mas não na 2FN. O atributo não primário ENOME viola a 2FN em razão da DF2, da mesma forma que os atributos não primários PNUMERO e PLOCALIZACAO em relação a DF3. As dependências funcionais DF2 e DF3 fazem ENOME, PNUMERO e PLOCALIZACAO parcialmente dependentes da chave primária {SSN, PNUMERO} de EMP_PROJ, violando, assim, o teste da 2FN.

Se um esquema de relação não estiver na 2FN, poderá ser 'normalizado na segunda forma', ou será feita 'normalização 2FN', por meio da criação de várias relações na 2FN nas quais os atributos não primários só estarão associados à parte da chave primária com a qual possuem dependência funcional total. As dependências funcionais DF1, DF2 e DF3 da Figura 10.3b conduzem, conseqüentemente, à decomposição de EMP_PROJ nos três esquemas de relação EP1, EP2 e EP3, mostrados na Figura 10.10a, cada um deles na 2FN.

10.3.6 Terceira Forma Normal

A **terceira forma normal (3FN)** está baseada no conceito de *dependência transitiva*. Uma dependência funcional $X \rightarrow Y$, em um esquema de relação R, será uma **dependência transitiva** se existir um conjunto de atributos Z que não é nem uma chave candidata nem um subconjunto de qualquer chave de R, e ambas $X \rightarrow Z$ e $Z \rightarrow Y$ forem asseguradas. A dependência SSN \rightarrow DGERSSN é transitiva para DNUMERO em EMP_DEPT da Figura 10.3a porque ambas as dependências SSN \rightarrow DNUMERO e DNUMERO \rightarrow DGERSSN são asseguradas e DNUMERO não é nem uma chave primária nem um subconjunto da chave de EMP_DEPT. Intuitivamente, podemos ver que a dependência de DGERSSN para DNUMERO é indesejável em EMP_DEPT, uma vez que DNUMERO não é chave de EMP_DEPT.

14 Esta é a definição geral de dependência transitiva. Por estarmos preocupados apenas com as chaves primárias nesta seção, permitiremos as dependências transitivas quando X for chave primária e Z, uma chave (ou um subconjunto da) candidata.

(a)

SSN	EMP_ P DNUMERO	HORAS	ENOME	PNOME	PLOCALIZACAO
DF1		u	t	t	
DF2					

DF3

t

y

NORMALIZAÇÃO 2FN

EP1

EP2

EP3

SSN	PNUMERO	HORAS
FD1	i	i
SSN	ENOME	
FD2	u	
PNUMERO	PNOME	PLOCALIZACAO
FD3	n	fr

(b)

EMP_DEPT

ENOME	SSN	DATANASC	ENDEREÇO	DNUMERO	DNOME	DGERSSN
i	i	I	A	h		

ED1

NORMALIZAÇÃO 3FN

ED2

ENOME	SSN	DATANASC	ENDEREÇO	DNUMERO
-------	-----	----------	----------	---------

i	i	i		
DNUMERO	DNOME	DGERSSN		
	i	i	t	

FIGURA 10.10 Normalização na 2FN e na 3FN. (a) Normalização EMP_PROJ em relações na 2FN. (b) Normalização EMP_DEPT em relações na 3FN.

Definição. De acordo com a definição original de Codd, um esquema de relação R está na 3FN se satisfizer a 2FN e se nenhum atributo não primário de R for transitivamente dependente da chave primária.

O esquema de relação EMP_DEPT da Figura 10.3a está na 2FN, uma vez que não existe nenhuma dependência parcial em uma chave. Porém, EMP_DEPT não está na 3FN em virtude da dependência transitiva de DGERSSN (e também de DNOME) para SSN via DNUMERO. Podemos normalizar EMP_DEPT decompondo-a nos dois esquemas de relação na 3FN, ED1 e ED2, mostrados na Figura 10.10b. Intuitivamente, vemos que ED1 e ED2 representam fatos independentes das entidades empregados e departamentos. Uma operação de NATURAL JOIN em ED1 e ED2 recuperará a relação original EMP_DEPT sem gerar tuplas ilegítimas.

Intuitivamente, podemos notar que qualquer dependência funcional, na qual o lado à esquerda for parte (um subconjunto) da chave primária ou qualquer dependência funcional na qual o lado esquerdo for um atributo não chave, é uma DF 'problemática'. As normalizações 2FN e 3FN removem esses problemas de DFs por meio da decomposição da relação original em relações novas. Em termos de processo de normalização, não é necessário remover as dependências parciais antes das dependências transitivas, mas historicamente a 3FN foi definida supondo-se que a relação já tivesse sido testada para a 2FN, antes de ser testada para 3FN. A Tabela 10.1 apresenta um resumo informal das três formas normais baseadas nas chaves primárias, os testes aplicados em cada caso e o correspondente 'remédio' ou a normalização executada para chegar à forma normal.

10.4 DEFINIÇÕES GERAIS DA SEGUNDA E DA TERCEIRA FORMAS NORMAL

Em geral, queremos projetar nossos esquemas de relação de forma que eles não tenham dependências parciais nem transitivas porque esses tipos de dependências geram as anomalias de atualização discutidas na Seção 10.1.2. Os passos para a normalização

10.4 Definições Gerais da Segunda e da Terceira Formas Normal 229

de relações na 3FN que discutimos até agora desaprovam as dependências parciais e transitivas na *chave primária*. Essas definições, porém, não levam em conta as demais chaves candidatas da relação, se houver alguma. Nesta seção damos definições mais gerais da 2FN e da 3FN, que consideram *todas* as chaves candidatas de uma relação. Observe que isso não afeta a definição da 1FN, uma vez que ela é independente de chaves e dependências funcionais. Como definição geral de **atributo primário**, será considerado como primário um atributo que faça parte de *qualquer chave candidata*. Serão consideradas agora as dependências funcionais parciais e totais, e as dependências transitivas com respeito a *todas as chaves candidatas* de uma relação.

TABELA 10.1 Resumo das Formas Normais Baseadas em Chaves Primárias e na Normalização Correspondente

TESTE PARA A FORMA NORMAL

Primeira (1FN) Segunda (2FN)

Terceira (3FN)

A relação não deve conter os atributos não atômicos ou as relações aninhadas (só conter os atributos atômicos).

Para as relações que possuam chaves primárias com vários atributos, nenhum atributo externo à chave deve ser funcionalmente dependente de parte da chave primária.

As relações não devem ter atributos que não pertençam a uma chave, funcionalmente determinados por outro atributo que também não pertença a uma chave (ou por um conjunto de atributos não-chave). Isto é, não deve haver dependência transitiva entre um atributo não-chave e uma chave primária.

AÇÕES (NORMALIZAÇÃO)

Formar uma nova relação para cada atributo não atômico ou para cada relação aninhada.

Decompor e montar uma nova relação para cada chave parcial com seu(s) atributo(s) dependente(s). Assegurar-se de que manteve a relação com a chave primária original e com todos os atributos que possuam dependência funcional total com ela.

Decompor e montar uma relação que contenha o(s) atributo(s) não-chave que determina(m) funcionalmente o(s) outro(s) atributo(s).

10.4.1 Definição Geral da Segunda Forma Normal

Definição. Um esquema de relação R está na segunda **forma** normal (2FN) se cada atributo não primário A de R não for parcialmente dependente de *nenhuma* chave de R.

O teste para 2FN verifica se os atributos do lado esquerdo das dependências funcionais *fazem parte* da chave primária. Se a chave primária contiver um único atributo, não há necessidade do teste. Considere o esquema da relação LOTES, mostrado na Figura 10.11a, que descreve os lotes de terra à venda em vários municípios de um estado. Suponha que existam duas chaves candidatas: NUM_ID_PROPRIEDADE e {MUNICIPIO_NOME, NUM_LOTE}, isto é, os números dos lotes são diferentes apenas dentro de cada município, mas os números NUM_ID_PROPRIEDADE são diferentes em qualquer município, para todo o estado.

Com base nas duas chaves candidatas NUM_ID_PROPRIEDADE e {MUNICIPIO_NOME, NUM_LOTE}, sabemos que as dependências funcionais DF1 e DF2 da Figura 10.11a estão asseguradas. Escolhemos NUM_ID_PROPRIEDADE como a chave primária, assim ela é sublinhada na Figura 10.11a, embora nenhuma consideração especial tenha sido feita a essa chave em relação às demais chaves candidatas. Suponha que as duas dependências funcionais a seguir sejam determinadas para LOTES:

DF3: MUNICIPIO_NOME -> IMPOSTO DF4: ÁREA -> PREÇO

Colocando em palavras, a dependência DF3 diz que a taxa de imposto é fixa para um determinado município (ela não varia entre os lotes de um mesmo município), enquanto DF4 diz que o preço de um lote é determinado por sua área, independentemente do município a que ele pertença (pressuponha que este seja o preço do lote para efeito de impostos).

15 Essa definição pode ser redeclarada da seguinte forma: um esquema de relação R está na 2FN se todo atributo não primário A de R possuir dependência funcional total de *cada* chave de R.

230 Capítulo 10 Dependência Funcional e Normalização em um Banco de Dados Relacional

(a) LOTES

NUM_ID_PROPRIEDADE		MUNICIPIO_NOME	NUM_LOTE	ÁREA	PREÇO	IMPOSTO
DF1	ii	n	A		\	' 1

A DF2 **t** **t** **1**

DF3 A

DF4

(b) LOTES 1

NUM_ID_PROPRIEDADE		MUNICIPIO_NOME	NUM_LOTE	ÁREA	PREÇO
--------------------	--	----------------	----------	------	-------

DF1 l. n n **t**

DF2 i n **i**

DF4 **t**

LOTES2

MUNICIPIO_NOME	IMPOSTO
----------------	---------

DF3 í ,

(c) LOTES 1A

LOTES1B

NUM_ID_PROPRIEDADE		MUNICIPIO_NOME	NUM_LOTE	ÁREA
--------------------	--	----------------	----------	------

DF1 i i . **J**

1 DF2 i
ÁREA PREÇO I
DF4 ii

(d)

LOTES

LOTES1A

LOTES1B

LOTES2

LOTES2

1FN

2FN

3FN

FIGURA 10.11 Normalização na 2FN e na 3FN. (a) A relação LOTES com dependências funcionais DF1 a DF4. (b) Decomposição para as relações na 2FN LOTES LOTES2. (c) Decomposição de LOTES para as relações na 3FN LOTES1A e LOTES1B. (d) Resumo do processo de normalização de LOTES.

O esquema de relação LOTES viola a definição geral da 2FN porque IMPOSTO é parcialmente dependente na chave candidata {MUNICIPIO_NOME, NUM_LOTE} em razão da DF3. Para a normalização na 2FN, LOTES foi decomposta nas duas relações LOTES1 e LOTES2, como mostrado na Figura 10.11b. Construímos LOTES1 removendo o atributo IMPOSTO, que viola a 2FN de LOTES, para colocá-lo com MUNICIPIO_MOME (à esquerda em DF3, que causa a dependência parcial) em outra relação LOTES2. ambos os LOTES1 e LOTES2 estão na 2FN. Note que DF4 não viola a 2FN e é mantido em LOTES 1.

10.4.2 Definição Geral da Terceira Forma Normal

Definição. Um esquema de relação R está na **terceira forma normal** (3FN) sempre que uma dependência funcional *não trivial* $X \rightarrow A$ for determinada em R, qualquer (a) X é uma superchave de R; ou (b) A é um atributo primário de R.

10.5 Forma Normal de Boyce-Codd 231

De acordo com essa definição, LOTES2 (Figura 10.11b) está na 3FN. Porém, DF4 em LOTES1 viola a 3FN porque ÁREA não é uma superchave, e PREÇO não é um atributo primário de LOTES1. Para normalizar LOTES1 na 3FN, decompomos LOTES1 nos esquemas de relação LOTES1A e LOTES1B, mostrados na Figura 10.11c. Construímos LOTES1A removendo o atributo PREÇO, que viola a 3FN em LOTES1, e o colocamos com ÁREA (lado esquerdo de DF4, que causa a dependência transitiva) em outra relação LOTES1B. LOTES1A e LOTES1B estão na 3FN.

Dois pontos valem uma nota nesse exemplo e na definição geral da 3FN:

- LOTES1 viola 3FN porque PREÇO é transitivamente dependente em cada uma das chaves candidatas de LOTES1 pelo atributo não primário ÁREA.
- Essa definição geral pode ser aplicada *diretamente* para verificar se um esquema de relação está na 3FN; *não* é preciso passar primeiro pela 2FN. Se aplicarmos a definição da 3FN anterior para LOTES, com as dependências DF1 a DF4, concluímos que *tanto* DF3 quanto DF4 violam a 3FN. Poderíamos decompor, assim, diretamente LOTES em LOTES1A, LOTES1B e LOTES2. Assim, as dependências transitivas e parciais que violam a 3FN podem ser removidas seguindo *qualquer ordem*.

10.4.3 Interpretação da Definição Geral da Terceira Forma Normal

Um esquema de relação R viola a definição geral de 3FN se alguma dependência funcional $X \rightarrow A$, assegurada em R, violar ambas as condições (a) e (b) da 3FN. Violar (b) significa que A é um atributo não primário. Violar (a) significa que X não é um superconjunto de nenhuma chave de R; conseqüentemente, X pode ser não primário ou pode ser um dado subconjunto de uma chave de R. Se X não for primário, temos uma dependência tipicamente transitiva que viola 3FN, enquanto se X for um dado subconjunto de uma chave de R, temos uma dependência parcial que viola a 3FN (e também a 2FN). Assim, chegamos a uma **definição geral alternativa para a 3FN**: um esquema de relação R está na 3FN se todo atributo não primário de R apresentar ambas as seguintes condições:

- Ter dependência funcional total para todas as chaves de R.
- Não ser transitivamente dependente de nenhuma chave de R.

10.5 FORMA NORMAL DE BOYCE-CODD

A **forma normal de Boyce-Codd (BCNF)** foi proposta como uma forma mais simples de 3FN, mas é considerada mais rígida que a 3FN. Isto é, toda relação na BCNF também está na 3FN, porém, uma relação na 3FN *não* está *necessariamente* na BCNF. Intuitivamente, podemos ver a necessidade de uma forma normal mais forte que a 3FN quando voltamos ao esquema da relação LOTES da Figura 10.11a, com suas quatro dependências funcionais, DF1 a DF4. Suponha que tenhamos milhares de lotes dentro de uma relação, mas que esses lotes sejam somente de dois municípios: Dekalb e Fulton. Suponha, também, que as áreas dos lotes no município de Dekalb sejam 0,5; 0,6; 0,7; 0,8; 0,9 e 1,0 acres, enquanto as áreas dos lotes no município de Fulton estejam entre 1,1; 1,2; . . . ; 1,9 e 2,0 acres. Em tal situação, teríamos uma dependência funcional adicional DF5: ÁREA \rightarrow MUNICIPIO_NOME. Se adicionarmos esta às outras dependências, o esquema de relação LOTES1A ainda estaria na 3FN, porque MUNICIPIO_NOME é um atributo primário.

A área de um lote que determina seu município, como especificada por DF5, pode ser representada por 16 tuplas em uma relação separada R(ÁREA, MUNICIPIO_NOME), uma vez que são somente 16 valores possíveis para ÁREA. Essa representação reduz a redundância de escrever a mesma informação nos milhares de tuplas em LOTES1A. A BCNF é uma *forma normal mais forte* que desaprova LOTES1A e sugere a necessidade de decompô-la.

Definição. Um esquema de relação R está na BCNF sempre que uma dependência funcional não trivial $X \rightarrow A$ for mantida em R, então X será uma superchave de R.

A definição formal da BCNF difere ligeiramente da definição da 3FN. A única diferença entre as definições da BCNF e da 3FN é a condição (b) da 3FN, que permite que A seja primário e não se aplica para a BCNF. Em nosso exemplo, DF5 viola BCNF em LOTES1A porque ÁREA não é uma superchave de LOTES1A. Observe que DF5 satisfaz a 3FN em LOTES1A porque MUNICIPIO_NOME é um atributo primário (condição b), mas essa condição não existe na definição da BCNF. Podemos decompor LOTES1A em duas relações na BCNF, LOTES1AXe LOTES1AY, como mostrado na Figura 10.12a. Essa decomposição elimina a dependência funcional DF2, pois seus atributos já não coexistem na mesma relação depois da decomposição.

Na prática, a maioria dos esquemas de relação que está na 3FN também está na BCNF. Somente quando, em um esquema de relação R, assegurada $X \rightarrow A$, X não for uma superchave e A for um atributo primário, R estará na 3FN, mas não na BCNF.

232 Capítulo 10 Dependência Funcional e Normalização em um Banco de Dados Relacional

O esquema de relação R mostrado na Figura 10.12b ilustra o caso geral dessa relação. De maneira ideal, os projetos de um banco de dados relacional devem se esforçar para alcançar a BCNF ou a 3FN em todo esquema de relação. Uma normalização somente até a 1FN ou a 2FN não é considerada adequada, uma vez que historicamente elas foram desenvolvidas como etapas intermediárias para a 3FN e para a BCNF.

(a)

LOTES1A

NUM_PROPRIEDADE	MUNICÍPIO NOME	NUMJ.OTE	ÁREA
DF1	il u		
DF2	il		
DF5 T_			

LOTES1AX

XX

Normalização BCNF

LOTES1AY

NUM_PROPRIEDADE	ÁREA	NUM_LOTE

ÁREA

MUNICÍPIO_NOME

(b)

DF1

i

DF2

FIGURA 10.12 Forma normal de Boyce-Codd. (a) Normalização BCNF de LOTES1A com a dependência funcional DF2 eliminada na decomposição, (b) Uma relação A esquemática com DFs; ela está na 3FN, mas não na BCNF.

Como outro exemplo, considere a Figura 10.13, que mostra a relação ENSINA com as seguintes dependências: DF1: {ALUNO, CURSO} → INSTRUTOR DF2:¹⁶ INSTRUTOR → CURSO

ENSINA

ALUNO	CURSO	INSTRUTOR
Narayan	Banco de dados	Mark
Smith	Banco de dados	Navathe
Smith	Sistemas operacionais	s Ammar
Smith	Teoria	Schulman
Wallace	Banco de dados	Mark
Wallace	Sistemas operacionais	s Ahamad
Wong	Banco de dados	Omiecinski
Zelaya	Banco de dados	Navathe

FIGURA 10.13 Uma relação ENSIN A que está na 3FN , mas não BCNF.

Observe que {ALUNO, CURSO} é uma chave candidata dessa relação e que as dependências mostradas seguem o padrão na Figura 10.12b, com ALUNO como A, CURSO como B e INSTRUTOR como C. Logo, essa relação está na 3FN, mas não na BCNF. A decomposição desse esquema de relação em dois esquemas não é direta porque é possível decompô-la em um dos três pares seguintes:

1. {ALUNO, INSTRUTOR} e {ALUNO, CURSO}.

16 Essa dependência, que implica 'cada instrutor ministrar um curso', é uma restrição dessa aplicação.

10.6 Resumo 233

2. {CURSO, INSTRUCTOR} e {CURSO, ALUNO}.

3. {INSTRUCTOR, CURSO} e {INSTRUCTOR, ALUNO}.

Todas as três decomposições 'suprimem' a dependência funcional DF1. A *decomposição desejável* dentre as apresentadas é a 3, porque não gerará tuplas ilegítimas depois de uma junção.

Um teste para determinar se uma decomposição é não aditiva (sem perda, *lossless*) será discutida na Seção 11.1.4, na Propriedade LJ1. Em geral, uma relação, que não estiver na BCNF, deveria ser decomposta até alcançá-la, mesmo renunciando à preservação de todas as dependências nas relações decompostas, como é o caso desse exemplo. É isso que o Algoritmo 11.3 faz: ele poderia ser usado anteriormente para se chegar à terceira decomposição de ENSINA.

10.6 RESUMO

Neste capítulo discutimos várias das armadilhas do projeto de um banco de dados relacional quando se raciocina intuitivamente. Identificamos algumas das medidas informais que indicam se um esquema de relação é 'bom' ou 'ruim', e proporcionamos diretrizes informais para um bom projeto. Apresentamos, então, alguns conceitos formais que nos permitem construir projetos relacionais, pela forma *top-down*, analisando as relações individualmente. Definimos esse processo de projeto por análise e decomposição por meio da introdução do processo de normalização.

Discutimos os problemas de anomalias de atualização que acontecem quando as redundâncias estão presentes nas relações. As medidas de qualidade informais dos esquemas de relação denotam a simplicidade e a clareza da semântica dos atributos e a presença de poucos nulls nas extensões (estados) das relações. Uma boa decomposição também poderia evitar o problema de geração de tuplas ilegítimas, como resultado de uma operação.

Definimos o conceito de dependência funcional e discutimos algumas de suas propriedades. As dependências funcionais especificam as restrições semânticas entre os atributos de um esquema de relação. Mostramos como, a partir de um determinado conjunto de dependências funcionais, podem ser deduzidas as dependências adicionais usando um conjunto de regras de inferência. Definimos os conceitos de clausura e de cobertura relacionadas às dependências funcionais. Então definimos a cobertura mínima de um conjunto de dependências e fornecemos um algoritmo para calculá-la. Também mostramos como verificar se dois conjuntos de dependências funcionais são equivalentes.

Descrevemos, então, o processo de normalização para alcançar bons projetos a partir de testes nas relações para identificação de tipos de dependências funcionais 'problemáticas'. Fornecemos um processo de normalizações sucessivas com base na chave primária predefinida de cada relação, então relaxamos essa exigência e proporcionamos as definições mais genéricas da segunda forma normal (2FN) e da terceira forma normal (3FN), que levam em conta todas as chaves candidatas da relação. Apresentamos exemplos para ilustrar de que maneira, usando a definição geral da 3FN, um dado esquema de relação pode ser analisado e eventualmente decomposto para produzir um conjunto de relações na 3FN.

Finalmente, apresentamos a forma normal de Boyce-Codd (BCNF) e discutimos por que ela é uma 3FN forte. Também ilustramos como a decomposição de uma relação não-BCNF deve ser feita considerando-se a exigência de decomposição não aditiva.

O Capítulo 11 apresenta uma síntese, além de algoritmos de decomposição, para os projetos relacionais de um banco de dados com base em dependências funcionais. Em relação à decomposição, discutimos os conceitos *de junção sem perda* (não aditiva) e *preservação de dependência*, que são necessários para alguns desses algoritmos. Outros tópicos do Capítulo 11 incluem as dependências multivaloradas, as dependências de junção e as quarta e quinta formas normais, que levam essas dependências em consideração.

Questões para Revisão

10.1. Discuta a semântica de atributo como uma medida informal de valor para um esquema de relação.

10.2. Discuta as anomalias de inserção, exclusão e alteração. Por que elas são consideradas ruins? Ilustre com exemplos.

10.3. Por que os *nulls* deveriam, na medida do possível, ser evitados em uma relação? Discuta o problema das tuplas ilegítimas e como podemos preveni-las.

10.4. Estabeleça as diretrizes informais discutidas para o projeto de um esquema de relação. Ilustre como a violação dessas diretrizes pode ser prejudicial.

10.5. O que é uma dependência funcional? Quais são as possíveis fontes de informação que definem as dependências funcionais mantidas entre os atributos de um esquema de relação?

234 Capítulo 10 Dependência Funcional e Normalização em um Banco de Dados Relacional

- 10.6. Por que não podemos deduzir automaticamente uma dependência funcional a partir de um estado particular da relação?
- 10.7. Qual o papel das regras de inferência de Armstrong — as três regras de inferência IR1 a IR3 — no desenvolvimento da teoria do projeto relacional?
- 10.8. O que significa 'completeza' e 'robustez' nas regras de inferência de Armstrong?
- 10.9. O que significa clausura de um conjunto de dependências funcionais? Ilustre com um exemplo.
- 10.10. Quando dois conjuntos de dependências funcionais são equivalentes? Como podemos determinar essa equivalência?
- 10.11. O que é um conjunto mínimo de dependências funcionais? Todo conjunto de dependências tem um conjunto mínimo equivalente? Eles sempre serão únicos?
- 10.12. A que se refere o termo *relação não normalizada*? Como as formas normais se desenvolveram historicamente, da primeira forma normal até a forma normal Boyce-Codd?
- 10.13. Defina a primeira, a segunda e a terceira formas normais quando somente as chaves primárias são consideradas. Quais são as definições gerais da 2FN e da 3FN, que consideram todas as chaves de uma relação? Diferencie estas das que levam em conta somente as chaves primárias.
- 10.14. Que dependências indesejáveis são evitadas quando uma relação está na 2FN?
- 10.15. Que dependências indesejáveis são evitadas quando uma relação está na 3FN?
- 10.16. Defina a forma normal de Boyce-Codd. Como ela difere da 3FN? Por que é considerada uma forma mais forte da 3FN?

Exercícios

- 10.17. Suponha que tenhamos as seguintes exigências para um banco de dados de uma universidade que deseja controlar dados dos alunos:
- A universidade mantém o nome de cada aluno (ENOME), número (ENUM), número do seguro social (SSN), endereço atual (EENDA) e telefone (ETEL), endereço permanente (EENDP) e telefone (ETELP), data de nascimento (DATANASC), sexo (SEXO), turma (TURMA) (calouro, secundarista, ..., formando), departamento principal (CODDEPP), departamento secundário (CODDEPS — se houver) e formação (PROG) (graduação, especialização, ..., mestrado, doutorado). SSN e número têm valores diferentes para cada aluno.
 - Cada departamento é descrito por um nome (DNOME), código de departamento (DCOD), número de escritório (DESC), telefone do escritório (DFONE) e corpo acadêmico (DCA). Ambos, nome e código, têm valores diferentes para cada departamento.
 - Cada curso tem um nome de curso (CNAME), descrição (COESC), número de curso (CNUM), número de horas por semana (CARGAHORARIA), nível (NÍVEL) e departamento que oferece o curso (CDEPT). O número do curso é diferente para cada curso.
 - Cada disciplina tem um instrutor (INOME), semestre (SEMESTRE), ano (ANO), curso (DISCURSO) e número de disciplina (SECDISC). O número das disciplinas relaciona as disciplinas diferentes do mesmo curso que ocorrem em um mesmo semestre/ano; seus valores são 1, 2, 3, ..., até o número total de disciplinas ministrado em cada semestre.
 - A nota refere-se ao aluno (SSN), em uma disciplina em particular, e a uma nota (NOTA).
- Projete um esquema de um banco de dados relacional para essa aplicação de um banco de dados. Primeiro, mostre dependências funcionais que deveriam existir entre os atributos. Então projete os esquemas de relação para o banco de dados na 3FN ou na BCNF. Especifique os atributos-chave de cada relação. Avalie qualquer exigência não especificada e faça as suposições apropriadas para contemplá-las.
- 10.18. Prove ou refute as seguintes regras de inferência para as dependências funcionais. A prova pode ser feita pelo argumento ou usando as regras de inferência IR1 a IR3. A refutação deveria ser feita por meio de um exemplo de relação que satisfaça as condições e as dependências funcionais do lado esquerdo da regra de inferência, mas não satisfaça as dependências do lado direito.
- $\{W \rightarrow Y, X \rightarrow Z\} \not\models \{WX \rightarrow Y\}$
 - $\{X \rightarrow Y\} \not\models \{YDZMX \rightarrow Z\}$
 - $\{X \rightarrow Y, X \rightarrow W, WY \rightarrow Z\} \not\models \{X \rightarrow Z\}$
 - $\{XY \rightarrow Z, Y \rightarrow W\} \not\models \{XW \rightarrow Z\}$
 - $\{X \rightarrow Z, Y \rightarrow Z\} \not\models \{X \rightarrow Y\}$
 - $\{X \rightarrow Y, XYH \rightarrow Z\} \not\models \{X \rightarrow Z\}$
 - $\{X \rightarrow Y, Z \rightarrow W\} \not\models \{XZ \rightarrow YW\}$ h. $\{XY \rightarrow Z, Z \rightarrow X\} \not\models \{Z \rightarrow Y\}$
 - $\{X \rightarrow Y, Y \rightarrow Z\} \models \{X \rightarrow YZ\}$ j. $\{XY \rightarrow Z, Z \rightarrow Y\} \models \{X \rightarrow Y\}$

10.6 Resumo

235

10.19. Considere os dois conjuntos seguintes de dependências funcionais: $F = \{A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H\}$ e $G = \{A \rightarrow CD, E \rightarrow AH\}$. Verifique se eles são equivalentes.

10.20. Considere o esquema de relação EMP_DEPT da Figura 10.3a e o seguinte conjunto G de dependências funcionais em EMP_DEPT: $G = \{SSN \rightarrow \{ENOME, DATNASC, ENDEREÇO, DNUMERO\}, DNUMERO \rightarrow \{DNOME, DGERSSN\}\}$. Calcule as clausuras $\{SSN\}^+$ e $\{DNUMERO\}^+$ com respeito a G.

10.21. O conjunto de dependências funcionais G do Exercício 10.20 é mínimo? Se não for, tente achar um conjunto mínimo de dependências funcionais que sejam equivalentes a G. Prove que seu conjunto é equivalente a G.

10.22. Que anomalias de atualização podem ocorrer nas relações EMP_PROJ e EMP_DEPT das figuras 10.3 e 10.4?

10.23. Em que forma normal o esquema de relação LOTES, da Figura 10.1 Ia, está em relação às interpretações restritivas das formas normais que consideram *apenas a chave primária*⁷. Estariam na mesma forma normal se as definições gerais de forma normal fossem usadas?

10.24. Prove que qualquer esquema de relação com dois atributos está na BCNF.

10.25. Por que as tuplas ilegítimas aparecem no resultado da junção das relações EMP_PROJ1 e EMP_LOCS, da Figura 10.5 (resultado mostrado na Figura 10.6)?

10.26. Considere a relação universal $R = \{A, B, C, D, E, F, G, H, I, J\}$ e o conjunto de dependências funcionais $F = \{A, B \rightarrow C, \{A\} \rightarrow \{D, E\}, \{B\} \rightarrow \{F\}, \{F\} \rightarrow \{G, H\}, \{D\} \rightarrow \{I, J\}\}$. Qual é a chave de R? Decomponha R em relações na 2FN e, depois, na 3FN.

10.27. Repita o Exercício 10.26 para o seguinte conjunto de dependências funcionais $G = \{A, B \rightarrow C, \{B, D\} \rightarrow \{E, F\}, \{A, D\} \rightarrow \{G, H\}, \{A\} \rightarrow \{I\}, \{H\} \rightarrow \{J\}\}$.

10.28. Considere a seguinte relação:

B

NUM TUPLA

10

10

11

12 13

14

bl b2 b4 b3 bl b3

cl o2 cl c4

cl c4

-1 #2 #3 #4 #5 #6

a. Dada a extensão anterior (estado), qual das seguintes dependências *podem ser asseguradas* para a relação? Se a dependência não puder ser garantida, explique por que *especificando as tuplas que causam a violação*.

i. $A \rightarrow B$; ii. $B \rightarrow C$; iii. $C \rightarrow B$; iv. $B \rightarrow A$; v. $C \rightarrow A$

b. A relação anterior tem uma chave candidata em potencial? Se tiver, qual é? Se não tiver, por que não tem?

10.29. Considere uma relação R(A, B, C, D, E) com as dependências seguintes:

$AB \rightarrow C, CD \rightarrow E, DE \rightarrow B$

AB é uma chave candidata dessa relação? Se não, ABD é? Explique sua resposta.

10.30. Considere a relação R que tem atributos que controlam os programas dos cursos e disciplinas em uma universidade; $R = \{\text{CursoNr}, \text{DiscNr}, \text{DeptOferece}, \text{Credito-Horas}, \text{CursoNivel}, \text{InstrutorSSN}, \text{Semestre}, \text{Ano}, \text{Dias_Horas}, \text{SalaNr}, \text{NrAluno}\}$. Suponha que as seguintes dependências funcionais estejam asseguradas em R:

$\{\text{CursoNr}\} \rightarrow \{\text{DeptOferece}, \text{Credito-Horas}, \text{CursoNivel}\}$

$\{\text{CursoNr}, \text{DiscNr}, \text{Semestre}, \text{Ano}\} \rightarrow \{\text{Dias_Horas}, \text{SalaNr}, \text{NrAluno}, \text{InstrutorSSN}\}$

$\{\text{SalaNr}, \text{Dias_Horas}, \text{Semestre}, \text{Ano}\} \rightarrow \{\text{InstrutorSSN}, \text{CursoNr}, \text{DiscNr}\}$

Tente determinar quais conjuntos de atributos formam chaves de R. Como você poderia normalizar essa relação?

10.31. Considere as seguintes relações do banco de dados de uma aplicação de processamento de pedidos da ABC Inc.:

236 Capítulo 10 Dependência Funcional e Normalização em um Banco de Dados Relacional

PEDIDO (PNr, Pdata, CustoNr, Soma_Total)

PEDIDO-ITEM (PNr, INr, Qtdd_Pedida, Total_Precio, Desconto%)

Presuma que cada artigo tenha um desconto diferente. O TOTAL_PRECO refere-se a um artigo, PDATA é a data na qual o pedido foi feito, e a SOMA_TOTAL é o valor do pedido. Se aplicarmos uma junção natural das relações PEDIDOITEM e PEDIDO nesse banco de dados, como será o esquema de relação resultante? Qual será sua chave? Mostre as DFs na relação resultante. Está na 2FN? Está na 3FN? Por que sim ou por que não? (Comente suas suposições, se você fizer alguma.) 10.32. Considere a seguinte relação:

VENDA_CARRO (CarNr, Data_Venda, VendedorNr, Comissao%, Desconto)

Pressuponha que um carro possa ser vendido por diversos vendedores e conseqüentemente {CarroNr, VendedorNr} a chave primária. Outras dependências são:

Data_Venda -> Desconto

VendedorNr -> Comissao%

Baseado em uma dada chave primária, essa relação está na 1FN, na 2FN ou na 3FN? Porque sim ou por que não? Quais normalizações sucessivas você faria para chegar à normalização total? 10.33. Considere a seguinte relação referente à publicação de livros:

LIVRO (Titulo, NomeAutor, TipoLivro, Preço, AfiAutor, Editora)

AfiAutor refere-se à afiliação do autor. Suponha que existam as seguintes dependências:

Titulo -> Editora, TipoLivro TipoLivro -> Preço Autor -> AfiAutor

a. Em que forma normal está a relação? Explique sua resposta.

b. Aplique as normalizações até que não se possa decompor mais as relações. Explique cada decomposição.

Bibliografia Seleccionada

As dependências funcionais foram introduzidas originalmente por Codd (1970). As definições originais da primeira, da segunda e da terceira formas normais também foram estabelecidas por Codd (1972a), na qual pode ser achada uma discussão sobre as anomalias de atualização. A forma normal de Boyce-Codd foi definida em Codd (1974). A definição da terceira foi uma normal, alternativa à definição de BCNF, que apresentamos aqui, é dada por Ullman (1988). Ullman (1988), Maie (1983) e Atzeni e De Antonellis (1993) apresentam muitos dos teoremas e das provas relativos às dependências funcionais. Armstrong (1974) mostra a 'completeza' e a 'robustez' das regras de inferência IR1 a IR.3. As referências adicionais para teoria de projetos relacionais são fornecidas no Capítulo 11.

11

Algoritmos para Projeto de Banco de Dados Relacional e Demais Dependências

Neste capítulo descreveremos alguns dos algoritmos para projeto de banco de dados relacional que utilizam a teoria da dependência funcional e da normalização, bem como outros tipos de dependência. No Capítulo 10 apresentamos as duas principais abordagens para o projeto de banco de dados relacional. A primeira abordagem utiliza a técnica de **projeto top-down** (descendente), atualmente a mais utilizada no projeto de aplicações comerciais de banco de dados. Ela envolve o projeto de um esquema conceitual em um modelo de dados de alto nível, tal como o modelo EER, e o mapeamento do esquema conceitual em um conjunto de relações utilizando procedimentos como aqueles discutidos no Capítulo 7. A seguir, cada uma das relações é analisada tendo como base as dependências funcionais — e lhes são atribuídas chaves primárias. Por meio da aplicação do procedimento de normalização da Seção 10.3 podemos remover das relações quaisquer dependências parciais e transitivas remanescentes. Em algumas metodologias de projeto, essa análise é aplicada diretamente, durante o projeto conceitual, aos atributos dos tipos entidade e relacionamento. Neste caso, as dependências indesejáveis são descobertas durante o projeto conceitual, e os esquemas de relação resultantes dos procedimentos de mapeamento estariam automaticamente em formas normais mais altas, de modo que não haveria necessidade de normalização adicional.

A segunda abordagem utiliza a técnica de **projeto bottom-up** (ascendente), uma abordagem mais purista, que vê o projeto do esquema de banco de dados relacional estritamente em função das dependências funcionais e de outros tipos especificados nos atributos do banco de dados. Ela também é conhecida por síntese relacional. Após o projetista do banco de dados especificar as dependências, um **algoritmo de normalização** é aplicado para sintetizar os esquemas de relação. Cada esquema de relação individual deve possuir as medidas de boa qualidade associadas à 3FN, à BCNF (FNBC) ou a alguma forma normal de nível mais alto.

Neste capítulo descreveremos alguns desses algoritmos de normalização, bem como outros tipos de dependência. Também mostraremos com mais detalhes as duas propriedades desejáveis: a de junções não aditivas (sem perdas) e a da preservação de dependência. Geralmente os algoritmos de normalização começam sintetizando um esquema de relação gigante, chamado **relação universal**, que é uma relação teórica que inclui todos os atributos do banco de dados. Depois realizaremos a **decomposição** — divisão em esquemas menores de relação — até que ela não seja mais possível ou desejável, com base nas dependências funcionais e em outros tipos de dependências especificadas pelo projetista do banco de dados.

Primeiro descreveremos na Seção 11.1 as duas **propriedades de decomposições** desejáveis, a saber: a propriedade de preservação da dependência e a propriedade de junção sem perdas (ou não aditiva), ambas utilizadas pelos algoritmos de projeto para obter decomposições desejáveis. É importante observar que é *insuficiente* testar os esquemas de relação *de maneira independente uns dos outros* em relação à compatibilidade com as formas normais de mais alto nível, como a 2FN, a 3FN e a BCNF. As relações resultantes devem satisfazer coletivamente essas duas propriedades adicionais para que elas possam ser qualificadas como um bom projeto. A Seção 11.2 apresenta diversos algoritmos de normalização baseados apenas nas dependências funcionais que podem ser utilizadas para projetar esquemas na 3FN e na BCNF.

Depois apresentaremos outros tipos de dependência de dados, inclusive as dependências multivaloradas e as dependências funcionais. A presença dessas dependências leva à definição da quarta forma normal (4FN) e da quinta forma normal (5FN), respectivamente. Também definiremos as dependências de inclusão e as dependências *template* (de molde, que até o momento não levaram a nenhuma nova forma normal). Será também brevemente abordada a forma normal domínio-chave (*domain-key*) — DKNF ou FNDC — considerada a forma normal mais genérica. É possível pular algumas ou todas as seções entre 11.4, 11.5 e 11.6 para um curso introdutório de bancos de dados.

11.1 PROPRIEDADES DAS DECOMPOSIÇÕES RELACIONAIS

Na Seção 11.1.1 daremos exemplos para mostrar que olhar para uma relação *individual* para verificar se ela está em uma forma normal de mais alto nível não garante, por si só, um bom projeto; em vez disso, um *conjunto de relações* que formam o esquema do banco de dados relacional deverá possuir certas propriedades adicionais para garantir um bom projeto. Nas seções 11.1.2 a 11.1.5 veremos duas dessas propriedades: a propriedade de preservação da dependência e a propriedade de junção sem perda ou não aditiva. A Seção 11.1.4 apresentará as decomposições binárias e a Seção 11.1.5, as decomposições sucessivas de junção não aditiva.

11.1.1 Decomposição de Relação e Insuficiência das Formas Normais

Os algoritmos de projeto de banco de dados relacional que apresentaremos na Seção 11.2 começam com um único esquema de relação universal $R = \{A_1, A_2, \dots, A_n\}$, que inclui *todos* os atributos do banco de dados. Implicitamente, fazemos a **suposição** da relação universal, que diz que todo nome de atributo é único. O conjunto F das dependências funcionais, que deve valer para os atributos de R , é especificado pelos projetistas do banco de dados e será fornecido aos algoritmos de projeto. Por meio do uso de dependências funcionais, os algoritmos decompõem o esquema de relação universal R em um conjunto $D = \{R_1, R_2, \dots, R_m\}$ de esquemas de relação, que se tornarão o esquema do banco de dados relacional; D é uma decomposição de R . Devemos assegurar que cada atributo de R apareça em pelo menos um esquema de relação R_i na decomposição, de forma que nenhum atributo seja 'perdido'; formalmente, temos

$$\bigcup_{i=1}^m R_i = R$$

que é chamado de condição de preservação de **atributo** de uma decomposição.

Outro objetivo a alcançar é que cada relação individual R_i da decomposição D esteja na BCNF ou na 3FN. Entretanto essa condição, por si só, não é suficiente para garantir um bom projeto de banco de dados. Devemos considerar a decomposição D da relação universal como um todo, além de olhar para as relações individuais. Para ilustrar este ponto, considere a relação EMP_LOCS (ENOME, PLOCALIZACAO) da Figura 10.5, que está na 3FN e também na BCNF. De fato, nenhum esquema de relação contém apenas dois atributos está automaticamente na BCNF. Embora EMP_LOCS esteja na BCNF, ela ainda faz surgir tuplas espúrias quando participa de junções com EMP_PROJ (SSN, PNUMERO, HORAS, PJNOME, PLOCALIZACAO) que não estão na BCNF (veja o resultado de junção natural na Figura 10.6). Portanto, EMP_LOCS representa um esquema de relação particularmente ruim por causa da semântica envolvida, segundo a qual PLOCALIZACAO fornece a localização de *um dos projetos* nos quais um empregado trabalha, junção de EMP_PROJ com PROJETO (PNOME, PNUMERO, PLOCALIZACAO, DNUM) da Figura 10.2 — que está na BCNF — também faz surgir tuplas espúrias. Isso enfatiza a necessidade de outro critério que, com a 3FN ou com a BCNF, previna tais projetos ruins. Nas próximas três subseções, discutiremos tais condições adicionais que devem ser mantidas em uma decomposição D como um todo.

11.1.2 Propriedade de Preservação da Dependência de uma Decomposição

Seria útil se cada dependência funcional $X \rightarrow Y$, especificada em F , aparecesse diretamente em um dos esquemas de relação R_i da decomposição D , ou pudesse ser inferida a partir das dependências que apareçam em algum R_i . Informalmente, essa é a *condição de preservação da dependência*. Desejamos preservar a dependência porque cada dependência em F representa uma restrição do banco de dados. Se uma das dependências não for representada em alguma relação individual R_i da decomposição, não poderemos garantir essa restrição quando estivermos manipulando uma relação individualmente; em vez disso, devemos garantir que a restrição seja preservada.

Como exercício, o leitor poderá provar que essa sentença é verdadeira.

11.1 Propriedades das Decomposições Relacionais

239

veríamos fazer a junção de duas ou mais relações de uma decomposição e então verificar se a dependência funcional foi mantida no resultado da operação JOIN. Claramente, trata-se um procedimento ineficiente e impraticável.

Não é necessário que as dependências exatas especificadas em F apareçam nas relações individuais da decomposição D. É suficiente que a união das dependências mantidas nas relações individuais de D sejam equivalentes às de F. Agora definiremos esses conceitos mais formalmente.

Definição. Dado um conjunto de dependências F sobre R, a projeção de F em R_i , denotada por $TT_{R_i}(F)$, em que R_i é um subconjunto de R, é o conjunto de dependências $X \rightarrow Y$ em F, tal que os atributos em X e Y estejam todos contidos em R_i . Por isso, a projeção de F em cada esquema de relação R_i da decomposição D é o conjunto de dependências funcionais de F, a clausura de F, tal que todos os seus atributos, tanto os do lado esquerdo quanto os do lado direito, estejam em R_i . Dizemos que uma decomposição $D = \{R_1, R_2, \dots, R_m\}$ de R é **preservadora de dependência** em relação a F se a união das projeções de F em cada relação R_j de D for equivalente a F, ou seja,

$$((TT_{R_1}(F)) \cup \dots \cup (TT_{R_m}(F)))^+ = F^+$$

Se uma decomposição não for preservadora de dependência, algumas dependências serão **perdidas** na decomposição. Conforme mencionamos anteriormente, para verificar se uma dependência perdida foi mantida, devemos realizar o JOIN de duas ou mais relações da decomposição para obter uma relação que inclua todos os atributos, tanto do lado esquerdo quanto do lado direito da dependência perdida, e depois verificarmos se a dependência ocorre no resultado do JOIN — uma opção que não é prática.

Um exemplo de decomposição que não preserva dependências é mostrado na Figura 10.12a, na qual a dependência funcional DF2 é perdida quando LOTES 1A é decomposta em {LOTES IAX, LOTES IAY}. Entretanto, as decomposições da Figura 10.11 são preservadoras de dependência. De maneira similar, para o exemplo da Figura 10.13, independentemente de qual decomposição da relação ENSINA (ALUNO, CURSO, INSTRUTOR) seja escolhida entre as três fornecidas no texto, uma ou ambas as dependências originalmente presentes serão perdidas. Abaixo apresentamos uma proposição relacionada a essa propriedade sem fornecer a demonstração.

PROPOSIÇÃO 1

E sempre possível encontrar uma decomposição D preservadora de dependência em relação a F, tal que cada relação R_i de D esteja na 3FN.

Na Seção 11.2.1, descreveremos o Algoritmo 11.2, que cria uma decomposição $D = \{R_1, R_2, \dots, R_m\}$ preservadora de dependência para uma relação universal R, tendo como base um conjunto de dependências funcionais F, tal que cada R_i de D esteja na 3FN.

11.1.3 Propriedade de Junção sem Perdas (Não Aditiva) de uma Decomposição

Uma outra propriedade que uma decomposição D deveria possuir é a de junção sem perdas ou de junção não aditiva, que assegura que nenhuma tupla espúria seja gerada quando uma operação NATURAL JOIN (JUNÇÃO NATURAL) for aplicada às relações da decomposição. Já ilustramos esse problema na Seção 10.1.4 com o exemplo das figuras 10.5 e 10.6. Como essa é uma propriedade da decomposição de *esquemas* de relação, a condição de não haver nenhuma tupla espúria deverá valer para *todo estado de relação válido* — ou seja, para todo estado de relação que satisfaça as dependências de F. Por isso, a propriedade de junção sem perdas é sempre definida em relação a um conjunto específico de dependências F.

Definição. Formalmente, uma decomposição $D = \{R_1, R_2, \dots, R_m\}$ de R possui a **propriedade de junção sem perdas (não aditiva)** em relação ao conjunto de dependências F sobre R se, para *todo* estado de relação r de R que satisfaça F, o seguinte for verdadeiro, no qual * é o NATURAL JOIN de todas as relações em D:

$$*(TT_{R_1}(r), \dots, TT_{R_m}(r)) = r$$

A palavra perda em *sem perdas* se refere à *perda de informação*, e não à perda de tuplas. Se uma decomposição não tiver a propriedade de junção sem perdas, devemos obter tuplas espúrias adicionais após as operações PROJECT (TT — PROJEÇÃO) e NATURAL JOIN (*) serem aplicadas; essas tuplas adicionais representam informações errôneas. Preferimos o termo junção não aditiva porque ele descreve a situação com mais precisão. Se a propriedade for válida em uma decomposição, teremos a garantia de que nenhuma tupla espúria, que gere informações errôneas, será adicionada ao resultado depois que operações de projeção e junção natural forem aplicadas.

A decomposição de EMP_PROJ (SSN, PNUMERO, HORAS, ENOME, PNOME, PLOCALIZACAO) da Figura 10.3 em EMP_LOCS (ENOME, PLOCALIZACAO) EMP_PROJ1 (SSN, PNUMERO, HORAS, PNOME, PLOCALIZACAO) da Figura 10.5 obviamente não tem a propriedade de junção sem perdas, conforme ilustrado pela Figura 10.6. Utilizaremos um procedimento genérico para testar se qualquer decomposição D de uma relação para n relações é sem perdas (não aditiva) em relação a um dado conjunto F de dependências funcionais da relação; e será apresentado como o Algoritmo 11.1 abaixo. É possível aplicar um teste mais simples para verificar se a decomposição não aditiva para decomposições binárias; esse teste será descrito na Seção 11.1.4.

Algoritmo 11.1: Teste da Propriedade de Junção sem Perdas (Não Aditiva)

Entrada: uma relação universal R , uma decomposição $D = \{R_1, R_2, \dots, R_n\}$ de R , e um conjunto F de dependências funcionais

1. Criar uma matriz S inicial com uma linha i para cada relação R_i de D , e uma coluna j para cada atributo A_j de R
2. Atribuir $S(i, j) := b^i$ para todas as entradas da matriz.
(* cada b^i é um símbolo distinto associado aos índices (i, j) *)
3. Para cada linha i representando o esquema de relação R_i {para cada coluna j representando o atributo A_j
{se (relação R_i inclui o atributo A_j) então faça $S(i, j) := a^i$;}; (* cada a^i é um símbolo distinto associado ao índice (i) *)
4. Repita o seguinte laço até que uma *execução completa do laço* não resulte em mudanças em S {para cada dependência funcional $X \rightarrow Y$ de F

{para todas as linhas de S que tenham os mesmos símbolos nas colunas correspondentes aos atributos de X {faça os símbolos em cada coluna de forma que um atributo em Y seja o mesmo em todas essas linhas, conforme segue: Se qualquer uma das linhas possuir um símbolo ' a^i ' para a coluna, atribua às outras linhas o mesmo símbolo ' a^i ' < coluna. Se nenhum símbolo ' a^i ' existir para o atributo em qualquer das linhas, escolha um dos símbolos ' b^i ' que ap recem em uma das linhas para o atributo, e atribua, às outras linhas, o mesmo símbolo ' b^i ' na coluna ;};};

5. Se uma linha for feita inteiramente de símbolos V , então a decomposição possui a propriedade de junção sem perdas; caso contrário, não possui.

Dada uma relação R que é decomposta em um número de relações R_1, R_2, \dots, R_n , o Algoritmo 11.1 inicializa a mat: que consideramos ser algum estado de relação r de R . A linha i de S representa uma tupla t_i (correspondente à relação R_i), que possui símbolos ' a^i ' nas colunas que correspondem aos atributos de R_i , e símbolos ' b^i ' nas colunas remanescentes. Depois o algoritmo transforma as linhas dessa matriz (durante o laço do passo 4), de forma que elas representem as tuplas que satisfaçam todas as dependências funcionais de F . Ao final do passo 4, quaisquer duas linhas de S — que representam duas tuplas de r — que tenham os mesmos valores nos atributos X do lado esquerdo de uma dependência funcional $X \rightarrow Y$ de F também terão os mesmos valores nos atributos Y do lado direito. Pode ser demonstrado que, após a aplicação do laço do passo 4, se qualquer linha de S tiver com todos os símbolos ' a^i ', então a decomposição D possui a propriedade de junção sem perdas em relação a F .

Se, porém, não resultar nenhuma linha com todos os símbolos ' a^i ', D não satisfaz a propriedade de junção sem perdas. Nesse caso, o estado de relação r representado por S ao final do algoritmo será um exemplo de um estado de relação r de R que satisfaz as dependências de F , mas que não satisfaz a condição de junção sem perdas. Portanto, essa relação serve de **contra-exemplo** que prova que D não possui a propriedade de junção sem perdas em relação a F . Observe que os símbolos ' a^i ' e não têm nenhum significado ao final do algoritmo.

A Figura 11.1a mostra como aplicamos o Algoritmo 11.1 para a decomposição do esquema de relação EMP_PROJ da Figura 10.3b em dois esquemas de relação EMP_PROJ1 e EMP_LOCS da Figura 10.5a. O laço do passo 4 do algoritmo não pode trocar quaisquer símbolos V por símbolos ' a^i '; por isso a matriz S resultante não possui uma linha com todos os símbolos ' a^i ' portanto, a decomposição não possui a propriedade de junção sem perdas.

A Figura 11.1b mostra uma outra decomposição de EMP_PROJ (em EMP_PROJ1 e EMP_LOCS) que possui a propriedade de junção sem perdas, e a Figura 11.1c mostra como aplicamos o algoritmo àquela decomposição. Uma vez que uma linha consiste apenas de símbolos ' a^i ', sabemos que a decomposição possui a propriedade de junção sem perdas, e poderemos parar de aplicar as dependências funcionais (passo 4 do algoritmo) à matriz S .

11.1 Propriedades das Decomposições Relacionais

241

(a) $F = \{SSN, ENOME, PNUMERO, PNOME, PLOCALIZACAO, HORAS\}$ $\circ = \{R_1, R_2\}$ $f_1 = EMPJ-OCS = \{ENOME, PLOCALIZACAO\}$ $f_2 = EMP_PROJ = \{SSN, PNUMERO, HORAS, PNOME, PLOCALIZACAO\}$
 $F = \{SSN \rightarrow ENOME; PNUMERO \rightarrow \{PNOME, PLOCALIZACAO\}; \{SSN, PNUMERO\} \rightarrow HORAS\}$ SSN $ENOME$
 $PNUMERO$ $PNOME$ $PLOCALIZACAO$ $HORAS$

b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
a_1	b_{22}	a_3	a_4	a_5	a_6

(nenhuma alteração na matriz após a aplicação das dependências funcionais)

(b) **EMP**

PROJETO

TRABALHAREM

SSN	ENOME			
PNUMERO	PNOME	PLOCALIZACAO		
SSN	PNUMERO	HORAS		

(c) $f = \{SSN, ENOME, PNUMERO, PNOME, PLOCALIZACAO, HORAS\}$ $D = \{R_1, f_2, f_3\}$

$f_1 = EMP = \{SSN, ENOME\}$

$f_2 = PROJ = \{PNUMERO, PNOME, PLOCALIZACAO\}$ $f_3 = TRABALHA_EM = \{SSN, PNUMERO, HORAS\}$

$F = \{SSN \rightarrow ENOME; PNUMERO \rightarrow \{PNOME, PLOCALIZACAO\}; \{SSN, PNUMERO\} \rightarrow HORAS\}$

f_1

f_2

SSN	ENOME	PNUMERO	PNOME	PLOCALIZACAO	HORAS
a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

(matriz original S no início do algoritmo)

SSN	ENOME	PNUMERO	PNOME	PLOCALIZACAO	HORAS
a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
a_1	V^2	a_3	V	$\wedge I^5$	a_6

(matriz S após a aplicação das duas primeiras dependências funcionais -a última linha é repleta de símbolos "a", portanto, paramos)

FIGURA 11.1 Teste da junção sem perdas (não aditiva) para decomposições n-árias. (a) Caso 1: a decomposição de EMP_PROJ em EMP_PROJ1 e EMP_LOCS falha no teste, (b) Uma decomposição de EMP_PROJ que possui a propriedade de junção sem perdas, (c) Caso 2: a decomposição de EMP_PROJ em EMP, PROJETO e TRABALHA_EM satisfaz o teste.

11.1.4 Teste da Propriedade de Junção Não Aditiva para Decomposições Binárias

O Algoritmo 11.1 nos permite testar se uma determinada decomposição D, para n relações, obedece à propriedade de junção sem perdas em relação a um conjunto F de dependências funcionais. Há um caso especial de decomposição chamada **decomposição binária** — decomposição de uma relação R em duas relações. Fornecemos um teste mais fácil de ser aplicado que o Algoritmo 11.1, porém, apesar de ser mais prático, ele é *limitado* apenas às decomposições binárias.

PROPRIEDADE LJ1 (TESTE DE JUNÇÃO SEM PERDAS PARA DECOMPOSIÇÕES BINÁRIAS)

Uma decomposição $D = \{R_1, R_2\}$ de R possui a propriedade de junção sem perdas (não aditiva), em relação a um conjunto F de dependências funcionais para R, se e somente se:

242 Capítulo 11 Algoritmos para Projeto de Banco de Dados Relacional e Demais Dependências

- A DF $((R_i \text{ PI } R_2) \rightarrow (R_j - R_2))$ estiver em F^+ , ou
- A DF $((R_i \text{ D } R_2) \rightarrow (R_2 - R_i))$ estiver em F^+

Você poderá verificar se essa propriedade é assegurada em relação aos nossos exemplos informais de normalizações sucessivas das seções 10.3 e 10.4-

11.1.5 Decomposições Sucessivas de Junção sem Perdas (Não Aditiva)

Vimos a decomposição sucessiva de relações durante o processo para a segunda e a terceira normalizações nas seções 10.3 e 10.4. Para verificar se essas decomposições não são aditivas, precisaremos assegurar uma outra propriedade, conforme exposto na Proposição 2.

PROPOSIÇÃO 2 (Preservação da Propriedade Não Aditiva em Decomposições Sucessivas)

Se uma decomposição $D = \{R_1, R_2, \dots, R_m\}$ de R possui a propriedade de junção não aditiva (sem perdas) em relação a um conjunto F de dependências funcionais sobre R , e se uma decomposição $D_i = \{Q_i, Q_2, \dots, Q_j\}$ de R_i possui a propriedade de junção não aditiva em relação à projeção de F sobre R_i , então a decomposição $D_2 = \{R_j, R_2, \dots, R_i, Q_j, Q_2, \dots, Q^A, R_{i+1}, \dots, R_m\}$ de R possui a propriedade de junção não aditiva em relação a F .

11.2 ALGORITMOS PARA O PROJETO DO ESQUEMA DE BANCO DE DADOS RELACIONAL.

Agora forneceremos três algoritmos para a criação de uma decomposição relacional. Cada algoritmo possui propriedades específicas, conforme discutiremos abaixo.

11.2.1 Decomposição Preservadora de Dependências para Esquemas na 3FN

O Algoritmo 11.2 cria uma decomposição preservadora de dependências $D = \{R_1, R_2, \dots, R_m\}$ de uma relação universal R baseada em um conjunto F de dependências funcionais, tal que cada R_i de D esteja na 3FN. Isso garante apenas a propriedade de preservação de dependências, mas *não* garante a propriedade de junção sem perdas. O primeiro passo do Algoritmo 11.2 é encontrar uma cobertura mínima G para F ; o Algoritmo 10.2 pode ser usado neste passo.

Algoritmo 11.2: Síntese Relacional para 3FN com a Preservação de Dependências

Entrada: Uma relação universal R e um conjunto F de dependências funcionais para os atributos de R .

1. Encontrar uma cobertura mínima G para F (usar o Algoritmo 10.2);
2. Para cada A_j do lado esquerdo de uma dependência funcional que aparece em G , criar um esquema de relação em D com atributos $\{X \cup \{A_j \cup \{A_2\} \dots \cup \{A_k\}\}$, no qual $X \rightarrow A_1$; $X \rightarrow A_2, \dots$, $X \rightarrow A_k$ são as únicas dependências em G , com X do lado esquerdo (X é a chave dessa relação);
3. Colocar quaisquer atributos remanescentes (que não tenham sido colocados em nenhuma relação) em um único esquema de relação para garantir a propriedade de preservação de atributo.

PROPOSIÇÃO 3

Todo esquema de relação criado pelo Algoritmo 11.2 está em 3FN. (Não forneceremos uma demonstração formal aqui; a demonstração depende de G ser um conjunto mínimo de dependências.)

É óbvio que todas as dependências de G são preservadas pelo algoritmo porque cada dependência aparece em uma das relações R_i da decomposição D . Uma vez que G é equivalente a F , todas as dependências de F ou são preservadas diretamente na decomposição ou são derivadas utilizando-se as regras de inferência da Seção 10.2.2 a partir daquelas nas relações resultantes, assim garantindo a propriedade da preservação das dependências. O Algoritmo 11.2 é chamado de algoritmo de síntese relacional porque cada esquema de relação R_i da decomposição é sintetizado (construído) a partir do conjunto de dependências funcionais de G com o mesmo lado esquerdo X .

2 Veja uma prova em Maier (1983) ou Ullman (1982).

11.2.2 Decomposição de Junção sem Perdas (Não Aditivas) para Esquemas na BCNF (FNBC)

O próximo algoritmo decompõe um esquema de relação universal $R = \{A_1, A_2, \dots, A_n\}$ em uma decomposição $D = \{R_1, R_2, \dots, R_m\}$, tal que cada R_i esteja na BCNF (FNBC — Forma Normal de Boyce-Codd), e a decomposição D possua a propriedade de junção sem perdas em relação a R . O Algoritmo 11.3 utiliza a Propriedade LJ1 e a Proposição 2 (preservação da propriedade não aditiva em decomposições sucessivas) para criar uma decomposição de junção não aditiva $D = \{R_1, R_2, \dots, R_m\}$ de uma relação universal R baseada em um conjunto F de dependências funcionais, tal que cada R_i de D esteja na BCNF.

Algoritmo 10.3: Decomposição Relacional para BCNF com Propriedade de Junção Não Aditiva **Entrada:** Uma relação universal R e um conjunto F de dependências funcionais para os atributos de R .

1. Inicializar $D := \{R\}$;

2. Enquanto existir um esquema de relação Q em D que não esteja na BCNF faça **I**
 escolha um esquema de relação Q em D que não esteja na BCNF; encontre uma dependência funcional $X \rightarrow Y$ de Q que viole a BCNF; substitua Q em D por dois esquemas de relação $(Q - Y)$ e $(X \cup Y)$;};

Cada vez que passarmos pelo laço do Algoritmo 11.3, decomporíamos um esquema de relação Q que não esteja na BCNF em dois esquemas de relação. De acordo com a Propriedade LJ1 para decomposições binárias e com a Proposição 2, a decomposição D possui a propriedade de junção não aditiva. Ao final do algoritmo, todos os esquemas de relação em D estarão na BCNF. O leitor poderá verificar que o exemplo de normalização das figuras 10.11 e 10.12 segue basicamente esse algoritmo. As dependências funcionais DF3, DF4 e posteriormente DF5 violam a BCNF, assim, a relação LOTES é decomposta adequadamente em relações na BCNF — e então a decomposição satisfará a propriedade de junção não aditiva. De maneira similar, se aplicarmos o algoritmo ao esquema de relação ENSINA da Figura 10.13, ele será decomposto em ENSINAI (INSTRUTOR, ALUNO) e em ENSINA2 (INSTRUTOR, CURSO), porque a dependência DF2: INSTRUTOR \rightarrow CURSO viola a BCNF.

No passo 2 do Algoritmo 11.3, é necessário determinar se um esquema de relação Q está na BCNF ou não. Um método para fazê-lo é testar, para cada dependência funcional $X \rightarrow Y$ de Q , se X falha em incluir todos os atributos de Q , determinando desse modo se X é ou não uma (super)chave de Q . Uma outra técnica é baseada na observação de que, se o esquema de relação Q violar a BCNF, existe um par de atributos A e B em Q tal que $\{Q - \{A, B\}\} \rightarrow A$; por meio do cálculo da clausura $\{Q - \{A, B\}\}$ para cada par de atributos $\{A, B\}$ de Q , e da verificação de que a clausura inclui A (ou B), poderemos determinar se Q está na BCNF.

11.2.3 Decomposição Preservadora de Dependências e de Junção Não Aditiva (sem Perdas) para Esquemas na 3FN

Se desejarmos que uma decomposição possua a propriedade de junção não aditiva e preserve as dependências, teremos de nos contentar com esquemas de relação na 3FN em vez de na BCNF. Uma modificação simples no Algoritmo 11.2, mostrada como Algoritmo 11.4, resulta em uma decomposição D de R que faz o seguinte:

- Preserva dependências.
- Possui a propriedade de junção não aditiva.
- Faz com que cada esquema resultante da decomposição esteja na 3FN.

Algoritmo 11.4: Síntese Relacional para 3FN com Preservação de Dependências e Propriedade de Junção Não Aditiva (Sem Perdas)

Entrada: Uma relação universal R e um conjunto F de dependências funcionais para os atributos de R .

1. Encontrar uma cobertura mínima G para F (use o Algoritmo 10.2).
2. Para cada X do lado esquerdo de uma dependência funcional que aparecer em G , criar um esquema de relação em D com atributos $\{X \cup \{A_1 \cup \{A_2\} \dots \cup \{A_k\}\}$, onde $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$ são as únicas dependências em G com X do lado esquerdo (X é a chave dessa relação);
3. Se nenhum dos esquemas de relação de D contiver uma chave de R , então criar um esquema de relação a mais em D que contenha atributos que formem uma chave de R .

Capítulo 11 Algoritmos para Projeto de Banco de Dados Relacional e Demais Dependências

Pode ser demonstrado que uma decomposição formada a partir de um conjunto de esquemas de relação criados pelo algoritmo anterior é preservadora de dependências e possui a propriedade de junção não aditiva. Além disso, cada esquema da decomposição está na 3FN. Esse algoritmo é uma melhoria em relação ao Algoritmo 11.2, uma vez que o anterior garantia apenas a preservação da dependência.

O passo 3 do Algoritmo 11.4 envolve a identificação de uma chave K de R . O Algoritmo 11.4a pode ser usado para identificar uma chave K de R , baseando-se em um dado conjunto de dependências funcionais F . Começamos atribuindo para K todos os atributos de R , então vamos removendo um atributo por vez e verificando se os atributos remanescentes ainda formam uma superchave. Observe que o conjunto de dependências funcionais utilizado para determinar uma chave no Algoritmo 11.4a pode ser tanto F quanto G , uma vez que eles são equivalentes. Observe, também, que o Algoritmo 11.4a determina apenas *uma* chave entre todas as possíveis chaves candidatas de R ; a chave resultante depende da ordem na qual os atributos foram removidos de R no passo 2.

Algoritmo 11.4a: Encontrando uma Chave K para R Dado um Conjunto F de Dependências Funcionais Entrada: Uma relação universal R e um conjunto F de dependências funcionais para os atributos de R .

1. Inicializar $K := R$;
2. Para cada atributo A de K {calcular $(K - A)$ em relação a F ;
se $(K - A)$ contiver todos os atributos de R , então fazer $K := K - \{A\}$ };

E importante observar que a teoria de decomposições de junção não aditiva está baseada na suposição de que *nenhum valor null é permitido para os atributos de junção*. A próxima seção discutirá alguns dos problemas que valores *null* podem causar nas decomposições relacionais.

11.2.4 Problemas com Valores Null e Tuplas Dangling

Devemos considerar com cuidado os problemas associados a valores *null* no projeto de um esquema de banco de dados relacional. Ainda não existe teoria de projeto relacional plenamente satisfatória que trate valores *null*. Um problema ocorre quando algumas tuplas possuem valores *null* para atributos que serão utilizados para realizar a junção de relações individuais na decomposição. Para ilustrar, considere o banco de dados mostrado na Figura 11.2a, no qual duas relações, EMPREGADO e DEPARTAMENTO, são mostradas. As duas últimas tuplas de empregado — Berger e Benitez — representam empregados recentemente contratados que ainda não foram designados a um departamento (suponha que isso não viole nenhuma restrição de integridade). Agora suponha que queiramos recuperar uma lista de valores (ENOME, DNAME) para todos os empregados. Se aplicarmos a operação NATURAL JOIN em EMPREGADO e DEPARTAMENTO (Figura 11.2b), as duas tuplas mencionadas anteriormente *não* aparecerão no resultado.

A operação OUTERJOIN, discutida no Capítulo 6, pode tratar esse problema. Relembre que, se tomarmos o LEFTOUTER JOIN de EMPREGADO com DEPARTAMENTO, as tuplas de EMPREGADO que possuírem valor *null* para o atributo de junção ainda assim aparecerão no resultado, tendo sido reunidas a uma tupla 'imaginária' de DEPARTAMENTO, que possui valores *null* para todos os seus atributos. A Figura 11.2c mostra o resultado.

Geralmente, sempre que um esquema de banco de dados relacional for projetado com duas ou mais relações interligadas por meio de chaves estrangeiras, um cuidado especial deverá ser tomado com valores *null* potenciais em chaves estrangeiras. Isso pode causar uma perda inesperada de informações em consultas que envolvam junções segundo tal chave estrangeira. Além disso, se valores *null* ocorrerem em outros atributos, tais como SALÁRIO, seus efeitos em funções predefinidas, tais como SUM (SOMA) e AVERAGE (MÉDIA), devem ser cuidadosamente avaliados.

Um problema relacionado a isso é o das *tuplas dangling* (pendentes), as quais podem ocorrer se levarmos uma decomposição longe demais. Suponha que decomponhamos a relação EMPREGADO da Figura 11.2a em EMPREGADO_1 e EMPREGADO_2, mostrados nas figuras 11.3a e 11.3b. Se aplicarmos a operação NATURAL JOIN em EMPREGADO_1 e EMPREGADO_2, obteremos a relação EMPREGADO original. Entretanto, poderíamos ter usado a representação alternativa, mostrada na Figura 11.3c, na qual *não se inclui uma tupla* em EMPREGADO_3 quando o empregado não tiver sido designado a um departamento (em vez de incluir uma tupla com valor *null* para DNUM, como em EMPREGADO_2). Se usarmos EMPREGADO_3 em vez de EMPREGADO_2 e aplicarmos uma NATURAL JOIN em

3 O passo 3 do Algoritmo 11.2 não é necessário no Algoritmo 11.4 para preservar atributos porque a chave incluirá quaisquer atributos não posicionados; esses são os atributos que não participam de nenhuma dependência funcional.

4 Isso às vezes acontece quando aplicamos a fragmentação vertical em uma relação no contexto de um banco de dados distribuído (Capítulo 25).

11.2 Algoritmos para o Projeto do Esquema de Banco de Dados Relacional

245

EMPREGADO_1 e EMPREGADO_3, as tuplas para Berger e Benitez não aparecerão no resultado; elas são chamadas de **tuplas *dangüing*** porque são representadas em apenas uma das duas relações que representam empregados, por isso são perdidas se aplicarmos uma operação (INNER) JOIN.

(a)

EMPREGADO

ENOME	SSN	DATANASC	ENDEREÇO	DNUM
-------	-----	----------	----------	------

Smith, John B. Wong, Franklin T. Zelaya, Alicia J. Wallace, Jennifer S. Narayan, Ramesh K. Engiish, Joyce A. Jabbar, Ahmad V. Borg,,James E. Berger, Anders C. Benitez, Carlos M.

123456789 333445555 999887777 987654321 666884444 453453453 987987987 888665555 999775555 888664444

1965-01-09 1955-12-08 1968-07-19 1941-06-20 1962-09-15 1972-07-31 1969-03-29 1937-11-10 1965-04-26 1963-01-09

731 Fondren, Houston, TX 638 Voss, Houston, TX 3321 Castle, Spring, TX 291 Berry, Bellaire, TX 975 Fire Oak, Humble, TX 5631 Rice, Houston, TX 980 Dallas, Houston, TX 450 Stone, Houston, TX 6530 Braes, Bellaire, TX 7654 Beech, Houston, TX

5 5 4 4 5 5 4 1 null null

DEPARTAMENTO

DNOME	DNUM	SNGERSSN
-------	------	----------

(b)

Pesquisa 5 333445555
 Administração 4 987654321
 Sede administrativa 1 888665555

ENOME	SSN	DATANASC	ENDEREÇO	DNUM	DNOME	SNGERSSN
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Pesquisa	333445555
Wong, Franklin T	333445555	1955-12-08	638 Voss, Houston, TX	5	Pesquisa	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administração	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administração	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Pesquisa	333445555
Engiish, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Pesquisa	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administração	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Sede Administrativa	888665555

(c)

ENOME	SSN	DATANASC	ENDEREÇO	DNUM	DNOME	SNGERSSN
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX			
Wong, Franklin T	333445555	1955-12-08	638 Voss, Houston, TX			
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX			
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX			
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX			
Engiish, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX			
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX			
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX			
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX			
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX			

FIGURA 11.2 Considerações ligadas a junções com valores *null*. (a) Algumas tuplas de EMPREGADO possuem valor *null* para o atributo de junção DNUM. (b) Resultado da aplicação de NATURAL JOIN nas relações EMPREGADO e DEPARTAMENTO, (c) Resultado da aplicação de LEFT OUTER JOIN em EMPREGADO e DEPARTAMENTO.

5 Pesquisa 333445555
 5 Pesquisa 333445555
 4 Administração 987654321
 4 Administração 987654321
 5 Pesquisa 333445555
 5 Pesquisa 333445555
 4 Administração 987654321
 1 SedeAdministrativa 888665555

null null null
 null null null

11.2.5 Discussão dos Algoritmos de Normalização

Um dos problemas com os algoritmos de normalização que descrevemos é que o projetista do banco de dados deve primeiro especificar todas as dependências funcionais relevantes entre os atributos do banco de dados. Isso não é uma tarefa simples para um grande banco de dados, com centenas de atributos. Uma falha na especificação de uma ou duas dependências importantes pode resultar em um projeto inadequado. Outro problema é que esses algoritmos, em geral, são *não determinísticos*. Por exemplo, os *algoritmos de síntese* (os algoritmos 11.2 e 11.4) exigem a especificação de uma cobertura mínima G para o con-

Capítulo 11 Algoritmos para Projeto de Banco de Dados Relacional e Demais Dependências

junto F de dependências funcionais. Como, em geral, pode haver muitas coberturas mínimas correspondentes para F, o algoritmo pode fornecer projetos diferentes, dependendo da cobertura mínima utilizada. Alguns desses projetos podem não ser desejáveis. Para verificar a violação da BCNF, o *algoritmo de decomposição* (Algoritmo 11.3) depende da ordem em que as dependências funcionais lhe forem fornecidas. Novamente, é possível que muitos projetos diferentes, correspondentes ao mesmo conjunto de dependências funcionais, possam surgir, dependendo da ordem na qual tais dependências são consideradas em relação à violação da BCNF. Alguns dos projetos podem ser muito bons, ao passo que outros podem ser inaceitáveis.

(a) EMPREGADO_1

ENOME	SSN	DATANASC	ENDEREÇO
Smith, John B.	123456789	1965-01-09	
Wong, Franklin T.	333445555	1955-12-08	
Zelaya, Alicia J.	999887777	1968-07-19	
Wallace, Jennifer S.	987654321	1941-06-20	
Narayan, Ramesh K.	666884444	1962-09-15	
English, Joyce A.	453453453	1972-07-31	
Jabbar, Ahmad V.	987987987	1969-03-29	
Borg, James E.	888665555	1937-11-10	
Berger, Anders C.	999775555	1965-04-26	
Benitez, Carlos M.	888664444	1963-01-09	

731 Fondren, Houston, TX 638 Voss, Houston, TX 3321 Castle, Spring, TX 291 Berry, Bellaire, TX 975 Fire Oak, Humble, TX 5631 Rice, Houston, TX 980 Dallas, Houston, TX 450 Stone, Houston, TX 6530 Braes, Bellaire, TX 7654 Beech, Houston, TX

(b) EMPREGADOR

(c) EMPREGADO 3

SSN	DNUM
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1
999775555	null
888664444	null

SSN	DNUM
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1

FIGURA 11.3 O problema da 'tupla *dangling*'. (a) A relação EMPREGADO_1 (inclui todos os atributos de EMPREGADO da Figura 11.2a, exceto DNUM), (b) A relação EMPREGADO_2 (inclui o atributo DNUM com valores *null*). (c) A relação EMPREGADO_3 (inclui o atributo DNUM, mas não inclui tuplas para as quais DNUM tem valor *null*).

Nem sempre é possível encontrar uma decomposição em esquemas de relação que preserve as dependências e que ainda permita que cada esquema de relação da decomposição esteja na BCNF (em vez de na 3FN, como no Algoritmo 11.4). Podemos verificar cada um dos esquemas de relação na 3FN de uma decomposição de modo a checar se ele satisfaz à BCNF. Se algum esquema de relação R_j não estiver na BCNF, poderemos optar por decompô-lo mais ou deixá-lo como está, na 3FN (com algumas possíveis anomalias de atualização). Esse fato, o de que nem sempre poderemos encontrar uma decomposição em esquemas de relação na BCNF que preserve as dependências, pode ser ilustrado pelos exemplos das figuras 10.12 e 10.13. As relações LOTES1A (Figura 10.12a) e ENSINA (Figura 10.13) não estão na BCNF, mas estão na 3FN. Qualquer tentativa de prosseguir a decomposição de ambas as relações resultará na perda da dependência DF2: {MUNICIPIO_NOME, NUM_LOTE} \rightarrow {NUM_PROPRIEDADE, ÁREA} em LOTES1A, ou na perda de DF1: {ALUNO, CURSO} \rightarrow INSTRUTOR em ENSINA.

A Tabela 11.1 resume as propriedades dos algoritmos discutidos neste capítulo até aqui.

TABELA 11.1 Resumo dos Algoritmos Apresentados nas seções 11.1 e 11.2

ALGORITMO

ENTRADA

SAÍDA

PROPRIEDADES/ OBJETIVO

COMENTÁRIOS

11.1

Uma decomposição D de R e um conjunto F de dependências funcionais

Resultado booleano: sim ou não para a propriedade de junção não aditiva

Teste para a decomposição de junção não aditiva

Veja um teste mais simples para decomposições binárias na Seção 11.1.4

11.3 Dependências Multivaloradas e a Quarta Forma Normal

247

TABELA 11.1 Resumo dos Algoritmos Apresentados nas seções 11.1 e 11.2. (*continuação*)

ALGORITMO

ENTRADA

SAÍDA

PROPRIEDADES/ OBJETIVO

COMENTÁRIOS

11.2

11.3

11.4

11.4a

Conjunto F de

dependências

funcionais

Conjunto F de

dependências

funcionais

Conjunto F de

dependências

funcionais

Esquema de relação R com um conjunto F de dependências funcionais

Um conjunto de relações na 3FN

Um conjunto de relações na BCNF

Um conjunto de relações na 3FN

Chave K de R

Preservação da dependência

Decomposição de junção não aditiva

Decomposição de junção não aditiva E decomposição com preservação de dependência

Para encontrar uma chave K (que é um subconjunto de R)

Não garante a satisfação da propriedade de junção sem perdas

Não garante a preservação da dependência

Pode não obter BCNF

A relação R inteira é sempre uma superchave default

11.3 DEPENDÊNCIAS MULTIVALORADAS E A QUARTA FORMA NORMAL

Até aqui discutimos apenas a dependência funcional, que, de longe, é o mais importante tipo de dependência na teoria de projeto de banco de dados relacional. Entretanto, em muitos casos, as relações têm restrições que não podem ser especificadas como dependências funcionais. Nesta seção discutiremos o conceito de *dependência multivalorada* (DMV — *multivalued dependency* MVD) e definiremos a *quarta forma normal*, com base nessa dependência. As dependências multivaloradas são consequência da primeira forma normal (1FN) (Seção 10.3.4), que não aceita que um atributo em uma tupla tenha um *conjunto de valores*. Se tivermos um ou mais atributos multivalorados *independentes* no mesmo esquema de relação, teremos o inconveniente de precisar repetir cada valor de um dos atributos com cada valor do outro atributo para manter o estado da relação consistente e a independência entre os atributos envolvidos. Essa dependência é especificada por uma dependência multivalorada. Por exemplo, considere a relação EMP mostrada na Figura 11.4a. Uma tupla nessa relação EMP representa o fato de que um empregado, cujo nome é ENOME, trabalha em um projeto cujo nome é PNome e possui um dependente cujo nome é DNome. Um empregado pode trabalhar em vários projetos e pode ter vários dependentes, e os projetos e os dependentes do empregado são independentes entre si. Para manter o estado da relação consistente, deveríamos ter uma tupla separada para representar cada combinação de um dependente do empregado e de um projeto do empregado. Essa restrição é especificada por uma dependência multivalorada na relação EMP. Informalmente, sempre que dois relacionamentos 1:N *independentes*, A:B e A:C, são misturados em uma mesma relação, uma DMV pode aparecer.

11.3.1 Definição Formal de Dependência Multivalorada

Definição. Uma dependência multivalorada $X \twoheadrightarrow Y$ especificada no esquema de relação R, no qual X e Y são ambos subconjuntos de R, especifica a seguinte restrição para qualquer estado r de R: se duas tuplas t_j e t_k existirem em r tal que $t_j[X] = t_k[X]$, então duas tuplas t_3 e t_4 também devem existir em r com as seguintes propriedades, onde usamos Z para denotar $(R - (X \cup Y))$:⁷

- $t_3[X] = t_4[X] = t_j[X] = t_k[X]$.
- $t_3[Y] = t_i[Y]$ e $t^4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ e $t_4[Z] = t_i[Z]$.

S

Em um diagrama ER, cada um poderia ser representado como um atributo multivalorado ou como um tipo entidade fraca (Capítulo 3).

⁶ As tuplas t_1 , t_2 , t_3 e t_4 não são necessariamente distintas.

⁷ Z é uma abreviação para os atributos remanescentes de R após os atributos em $(X \cup Y)$ serem removidos de R.

Sempre que $X \rightarrow Y$ ocorrer, dizemos que X **multidetermina** Y . Por causa da simetria na definição, sempre que $X \rightarrow Y$ ocorrer em R , $X \rightarrow Z$ também ocorre. Por isso, $X \rightarrow Y$ implica $X \rightarrow Z$, portanto, às vezes é escrito como $X \rightarrow Y \mid Z$.

(a)

EMP

ENOME	PNOME	DNOME

Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(b)

EMP_PROJETOS

ENOME	PNOME

EMP_DEPENDENTES

ENOME	DNOME

(c)	Smith Smith	X Y	Smith Smith	John Anna
-----	-------------	-----	-------------	-----------

FORNECE

FNOME	NOMEPECA	NOMEPROJ
Smith	Parafuso	ProjX
Smith	Porca	ProjY ProjY ProjZ
Adamsky	Prego	ProjX
Walton	Porca	
Adamsky	Prego	

Adamsky Smith	Parafuso Parafuso	ProjX ProjY
---------------	-------------------	-------------

(d) **R1**

FNOME	NOMEPECA

R2**R3**

FNOME	NOMEPROJ
NOMEPECA	NOMEPROJ

Smith	Parafuso	Smith	ProjX	Parafuso	ProjX
Smith	Porca	Smith	ProjY	Porca	ProjY
Adamsky	Parafuso	Adamsky	ProjY	Parafuso	ProjY
Walton	Porca	Walton	ProjZ	Porca	ProjZ
Adamsky	Prego	Adamsky	ProjX	Prego	ProjX

FIGURA 11.4 A quarta e a quinta formas normais, (a) A relação EMP com duas DMVs: $ENOME \rightarrow PNOME$ e $ENOME \rightarrow DNOME$. (b) A decomposição da relação EMP em duas relações na 4FN EMP_PROJETOS e EMP_DEPENDENTES. (c) A relação FORNECE sem DMV está na 4FN, mas não na 5FN se ela possuir a DJ(R1, R2, R3). (d) A decomposição da relação FORNECE nas relações na 5FN **R1**, **R2**, **R3**.

A definição formal específica que dado um valor de X em particular, o conjunto de valores de Y determinados por esse valor de X é completamente determinado apenas por X e *não depende* dos valores dos atributos Z remanescentes de R . Por isso, sempre que existirem duas tuplas que tenham valores de Y distintos, porém o mesmo valor para X , os valores de Y devem ser repetidos em tuplas separadas com *cada valor de Z distinto* que ocorra com o mesmo valor de X . Informalmente, isso corresponde a Y ser um atributo multivalorado das entidades representadas pelas tuplas de R .

Na Figura 11.4a, as DMVs $ENOME \rightarrow PNOME$ e $ENOME \rightarrow DNOME$ (ou $ENOME \rightarrow PNOME \mid DNOME$) ocorrem na relação EMP. O empregado com ENOME 'Smi th' trabalha nos projetos com PNOME 'X' e 'Y', e possui dois dependentes com DNOME 'John' e 'Anna'. Se armazenarmos apenas as duas primeiras tuplas em EMP ($\langle \text{'Smith', 'X', 'John'} \rangle$ e $\langle \text{'Smith', 'Y', 'Anna'} \rangle$) de maneira incorreta, mostraríamos associações entre o projeto 'X' e 'John' e entre o projeto 'Y' e 'Anna'; tais associações não deveriam ser obtidas porque tal significado não é pretendido nessa relação. Por isso devemos armazenar também as outras duas tuplas ($\langle \text{'Smi th', 'X', 'Anna'} \rangle$ e $\langle \text{'Smi th', 'Y', 'John'} \rangle$) para mostrar que $\{X, Y\}$ e $\{John, Anna\}$ são associados apenas com 'Smi th', ou seja, não há associação entre PNOME e DNOME — o que significa que os dois atributos são independentes.

Uma DMV $X \rightarrow Y$ em R é chamada de DMV **trivial** se (a) Y é um subconjunto de X ou (b) $X \cup Y = R$. Por exemplo, a relação EMP_PROJETOS da Figura 11.4b tem a DMV trivial $ENOME \rightarrow PNOME$. Uma DMV que não satisfaz (a) nem (b) é chamada de DMV **não trivial**.

trivial. Uma DMV trivial ocorrerá em *qualquer* estado de relação r de R ; ela é chamada de trivial porque não especifica nenhuma restrição importante ou significativa sobre R .

Se tivermos uma DMV não trivial em uma relação, poderíamos precisar repetir os valores de maneira redundante nas tu-plas. Na relação EMP da Figura 11.4a, os valores 'X' e 'Y' de PNAME são repetidos para cada valor de DNAME (ou, por simetria, os valores 'John' e 'Anna' de DNAME são repetidos com cada valor de PNAME). Tal redundância é obviamente indesejável. Entretanto, o esquema EMP está na BCNF porque *nenhuma* dependência funcional ocorre em EMP. Portanto, precisaremos definir uma quarta forma normal que é mais forte que a BCNF e que não permitirá esquemas de relação como o de EMP. Primeiro discutiremos algumas das propriedades de DMVs e consideraremos como elas estão relacionadas às dependências funcionais. Observe que as relações que contêm DMVs não triviais tendem a ser relações *alkey* (tudo é chave) — ou seja, sua chave é formada por todos os seus atributos tomados em conjunto.

11.3.2 Regras de Inferência para Dependências Funcionais e Multivaloradas

Da mesma forma que as dependências funcionais (DFs), foram desenvolvidas regras de inferência para dependências multivaloradas (DMS). Entretanto, é melhor desenvolver um quadro unificado que inclua tanto as DFs quanto as DMVs, de forma que ambos os tipos de restrições possam ser considerados juntos. As seguintes regras de inferência desde a IR1 até a IR8 formam um conjunto sólido e completo para inferir dependências funcionais e multivaloradas a partir de um dado conjunto de dependências. Suponha que todos os atributos estejam incluídos em um esquema de relação 'universal' $R = \{A_1, A_2, \dots, A_n\}$ e que X, Y, Z , e W são subconjuntos de R .

IR1 (regra reflexiva para DFs): Se $X \rightarrow Y$, então $X \rightarrow Y$. IR2 [regra do acréscimo (*augmentation*) para DFs]: $\{X \rightarrow Y\} \vdash XZ \rightarrow YZ$. IR3 (regra transitiva para DFs): $\{X \rightarrow Y, Y \rightarrow Z\} \vdash X \rightarrow Z$. IR4 (regra do complemento para DMVs): $\{X \rightarrow Y\} \vdash (R - (X \cup Y))$. IR5 [regra do acréscimo (*augmentation*) para DMVs]: Se $X \twoheadrightarrow Y \twoheadrightarrow Z$, então $WX \twoheadrightarrow YZ$. IR6 (regra transitiva para DMVs): $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \vdash X \twoheadrightarrow (Z - Y)$. IR7 (regra da replicação de DF para DMV): $\{X \rightarrow Y\} \vdash X \twoheadrightarrow Y$.

IR8 (regra da coalescência para DFs e DMVs): Se $X \rightarrow Y$ existir W com as propriedades (a) $W \setminus Y$ é vazio, (b) $W \twoheadrightarrow Z$, e (c) $Y \rightarrow Z$, então $X \rightarrow Z$.

De IR1 a IR3 são as regras de inferência de Armstrong para DFs sozinhas. De IR4 a IR6 são as regras pertinentes apenas às DMVs. A IR7 e a IR8 se referem às DFs e às DMVs. A IR7, em particular, diz que uma dependência funcional é um *caso especial* de uma dependência multivalorada, ou seja, toda DF é também uma DMV porque ela satisfaz a definição formal de uma DMV. Entretanto, essa equivalência é uma armadilha: uma DF $X \rightarrow Y$ é uma DMV $X \twoheadrightarrow Y$ com a *restrição adicional implícita* de que no máximo um valor de Y está associado a cada valor de X . Dado um conjunto F de dependências funcionais e multivaloradas especificadas em $R = (A_1, A_2, \dots, A_n)$, podemos usar de IR1 a IR8 para inferir o conjunto (completo) de todas as dependências (funcionais ou multivaloradas) F que ocorrem em todo estado de relação r de R que satisfizer F . Novamente, chamaremos F de **clausura** de F .

11.3.3 Quarta Forma Normal

Agora apresentaremos a definição da **quarta** forma normal (4FN), que é violada quando uma relação possuir dependências multivaloradas indesejáveis, podendo, por isso, ser usada para identificar e decompor tais relações.

Definição. Um esquema de relação R está na 4FN em relação a um conjunto F de dependências (que inclui dependências funcionais e dependências multivaloradas) se, para cada dependência multivalorada *não trivial* $X \twoheadrightarrow Y$ em F , X for uma superchave de R .

A relação EMP da Figura 11.4a não está na 4FN porque, nas DMVs não triviais $ENAME \twoheadrightarrow PNAME$ e $ENAME \twoheadrightarrow DNAME$, $ENAME$ não é uma superchave de EMP. Decompomos EMP em EMP_PROJETOS e EMP_DEPENDENTES, mostradas na Figura 11.4b. Ambas, EMP_PROJETOS e EMP_DEPENDENTES, estão na 4FN, porque as DMVs $ENAME \twoheadrightarrow PNAME$ em EMP_PROJETOS e $ENAME \twoheadrightarrow DNAME$ em EMP_DEPENDENTES são DMVs triviais. Nenhuma outra DMV não trivial ocorre em EMP_PROJETOS ou em EMP_DEPENDENTES. Tampouco nenhuma DF ocorre nesses esquemas de relação.

8 O conjunto de valores de Y , determinados por um valor de X , se restringe a ser um *conjunto singleton (monobloco)* com apenas um valor. Por isso, na prática, nunca vemos uma DF como uma DMV.

250 Capítulo 11 Algoritmos para Projeto de Banco de Dados Relacional e Demais Dependências

Para ilustrar a importância da 4FN, a Figura 11.5a mostra a relação EMP com um empregado a mais, 'Brown', que possui três dependentes ('Jim', 'Joan' e 'Bob') e trabalha em quatro projetos diferentes ('W', 'X', 'Y' e 'Z'). Há 16 tuplas em EMP na Figura 11.5a. Se fizermos a decomposição de EMP para EMP_PROJETOS e EMP_DEPENDENTES, conforme mostrado na Figura 11.5b, precisaremos armazenar um total de apenas onze tuplas em ambas as relações. A decomposição não apenas economizaria espaço de armazenamento, mas as anomalias de atualização associadas às dependências multivaloradas também seriam evitadas. Por exemplo, se Brown começar a trabalhar em um novo projeto P, deveremos incluir três tuplas em EMP — uma para cada dependente. Se esquecermos de incluir qualquer uma delas, a relação violaria a DMV e se tornaria inconsistente, uma vez que implicaria, de forma incorreta, um relacionamento entre projeto e dependente.

(b) EMP_PROJETOS

(a) EMP

ENOME	PNOME	DNOME
Smith	X	John

Smith	Y	Anna
Smith	X	Anna
Smith	Y	John
Brown	W	Jim
Brown	X	Jim
Brown	Y	Jim
Brown	Z	Jim
Brown	W	Joan
Brown	X	Joan
Brown	Y	Joan
Brown	Z	Joan
Brown	W	Bob
Brown	X	Bob
Brown	Y	Bob
Brown	Z	Bob

ENOME	PNOME
Smith	X

Smith	Y
Brown	W
Brown	X
Brown	Y
Brown	Z

EMP_DEPENDENTES

ENOME	DNOME
Smith	Anna

Smith	John
Brown	Jim
Brown	Joan
Brown	Bob

FIGURA 11.5 Decomposição de um estado de relação de EMP que não está na 4FN. (a) A relação EMP com tuplas adicionais. (b) Duas relações correspondentes, EMP_PROJETOS e EMP_DEPENDENTES, na 4FN.

Se a relação possuir DMVs não triviais, então as operações de inclusão, exclusão e atualização em tuplas únicas podem causar a modificação de outras tuplas, além daquela em questão. Se a atualização for tratada incorretamente, o significado da relação pode mudar. Entretanto, após a normalização para a 4NF, estas anomalias de atualização desaparecem. Por exemplo, para adicionar a informação de que Brown será designado ao projeto P, seria necessário incluir apenas uma única tupla na relação, na 4NF, EMP_PROJETOS.

A relação EMP da Figura 11.4a não está na 4FN porque representa dois relacionamentos 1:N independentes — um entre empregados e os projetos nos quais eles trabalham e o outro entre empregados e seus dependentes. Às vezes, temos um relacionamento entre três entidades que depende de todas as três entidades participantes, tal como a relação FORNECE mostrada na Figura 11.4c. (Por enquanto, considere apenas as tuplas da Figura 11.4c acima da linha pontilhada.) Nesse caso, uma tupla representa um fornecedor que abastece, para um projeto em particular, uma peça, portanto, não há DMVs triviais. A relação FORNECE já está na 4FN e não deve ser decomposta.

11.3.4 Decomposição de Junção sem Perdas (Não Aditiva) em Relações na 4FN

Sempre que fizermos a decomposição de um esquema de relação R em $R_1 = (X \cup Y)$ e $R_2 = (R - Y)$ baseado em uma DMV X-Y que ocorre em R, a decomposição tem a propriedade de junção não aditiva. Pode ser demonstrado que essa é uma condição necessária e suficiente para a decomposição de um esquema em dois esquemas que possuem a propriedade de junção não aditiva conforme se vê

na Propriedade LJ 1', que é uma generalização avançada da Propriedade LJ 1, apresentada anteriormente. A propriedade LJ1 tratava apenas de DFs, enquanto a LJ 1' trata de ambas, DFs e DMVs (lembre-se de que uma DF também é uma DMV)

PROPRIEDADE LJr

Os esquemas de relação R_1 e R_2 formam uma decomposição de junção não aditiva de R em relação a um conjunto F de dependências funcionais e multivaloradas se e somente se

$$(R_1 \bowtie R_2) \wedge (R_1 - R_2)$$

ou, por simetria, se e somente se

$$(R, nR_2) \wedge (R_2, R_1)$$

Podemos fazer uma pequena modificação no Algoritmo 11.3 para desenvolver o Algoritmo 11.5, que cria uma decomposição de junção não aditiva em esquemas de relação que estão na 4FN (em vez de na BCNF). Da mesma forma que no Algoritmo 11.3, o Algoritmo 11.5 *não* produz necessariamente uma decomposição que preserve as DFs.

Algoritmo 11.5: Decomposição Relacional para Relações na 4FN com a Propriedade de Junção Não Aditiva **Entrada:** Uma relação universal R e um conjunto F de dependências funcionais e multivaloradas.

1. Inicializar $D := \{ R \}$;
2. Enquanto existir um esquema de relação Q em D que não esteja na 4FN, fazer { escolha um esquema de relação Q em D que não esteja na 4FN; encontre uma DMV não trivial $X \rightarrow Y$ em Q que viole a 4FN; substitua Q em D por dois esquemas de relação $(Q-Y) \bowtie (X \cup Y)$; };

11.4 DEPENDÊNCIAS DE JUNÇÃO E A QUINTA FORMA NORMAL

Vimos que LJ1 e LJ1' dão condição para que um esquema de relação R seja decomposto em dois esquemas R_j e R_2 , e a decomposição terá a propriedade de junção não aditiva. Entretanto, em alguns casos pode não haver decomposição de junção não aditiva de R em *dois* esquemas de relação, mas pode haver uma decomposição de junção não aditiva (sem perdas) em *mais de dois* esquemas de relação. Além disso, pode não haver dependência funcional em R que viole qualquer forma normal até a BCNF, e pode não haver nenhuma outra DMV não trivial em R que viole a 4FN. Então recorremos a uma outra dependência, chamada *dependência de junção*, e se ela estiver presente, realizaremos uma *decomposição multivias* para a quinta forma normal (5FN). É importante observar que tal dependência é uma restrição semântica muito particular e muito difícil de ser detectada na prática, portanto, a normalização para a 5FN raramente é realizada na prática.

Definição. Uma **dependência de junção (Dj)**, denotada por $DJ(R_j, R_2, \dots, R_n)$, especificada em um esquema de relação R , especifica uma restrição nos estados r de R . A restrição diz que todo estado legal r de R deveria ter uma decomposição de junção não aditiva para R_j, R_2, \dots, R_n , ou seja, para todo r que tenhamos

$$*(\text{Tr}_{R_1}(r), \text{Tr}_{R_2}(r), \dots, \text{Tr}_{R_n}(r)) = r$$

Observe que uma DMV é um caso especial de DJ na qual $n = 2$. Ou seja, uma DJ denotada por $Dj(R_j, R_2)$ implica uma DMV $(R \wedge R^* \rightarrow (R_j - R_2))$ [ou, por simetria, $(R_1 - R_2) \rightarrow (R \wedge R^*)$]. Uma dependência de junção $Dj(R_1, R_2, \dots, R_n)$, especificada sobre um esquema de relação R , é uma DJ **trivial** se um dos esquemas de relação R_i de $DJ(R_j, R_2, \dots, R_n)$ for igual a R . Tal dependência é chamada de trivial porque possui a propriedade de junção não aditiva para qualquer estado de relação r de R , e por isso não especifica qualquer restrição em R . Agora podemos definir a quinta forma normal, que também é chamada de forma normal projeção-junção (*project-join*).

Definição. Um esquema de relação R está na **quinta forma normal (5FN)** [OU na **forma normal projeção-junção (PNJF — FNPj)**] em relação a um conjunto F de dependências funcionais, multivaloradas e de junção se, para cada dependência de junção não trivial $Dj(R_1, R_2, \dots, R_n)$ de F (ou seja, implicada por F), todo R_i for uma superchave de R .

Como exemplo de uma DJ, considere novamente a relação *all-key* FORNECE da Figura 11.4c. Suponha que a seguinte restrição adicional sempre seja mantida: toda vez que um fornecedor/fornecer a peça p , e um projeto j usar a peça p , e o fornecedor /fornecer *pelo menos uma* peça ao projeto j , então o fornecedor/também estará fornecendo a peça p para o projeto j . Essa restrição pode ser reescrita de outras maneiras e especifica uma dependência de junção $Dj(R_1, R_2, R_3)$ entre as três projeções $R_1(\text{FOME}, \text{NOMEPEÇA})$, $R_2(\text{FOME}, \text{PNOME})$ e $R_3(\text{NOMEPEÇA}, \text{PNOME})$ de FORNECE. Se essa restrição for assegurada, as tuplas abaixo da linha pontilhada na Figura 11.4c precisam existir em todo estado legal da relação FORNECE, que também deverá conter as tuplas acima da linha pontilhada. A Figura 11.4d mostra como a relação FORNECE *com a dependência de junção* é decomposta em três relações, R_1, R_2 e R_3 que estão, cada uma, na 5FN. Observe que a aplicação de uma junção natural a *duas quaisquer* dessas relações *produz tuplas espúrias*, mas a aplicação de uma junção natural, em todas *as três juntas*, não produzirá tuplas espúrias. O leitor pode verificar isso no exemplo de relação da Figura 11.4c e nas projeções na Figura 11.4d. Isso ocorre porque apenas a DJ existe, mas

normalização apresentado em Biskup *et al.* (1979). Tsou e Fischer (1982) fornecem um algoritmo polinomial (*polynomial-time*) para a decomposição BCNF.

A teoria da preservação de dependências e as junções sem perdas são dadas em Ullman (1988), na qual aparecem as provas para alguns dos algoritmos discutidos aqui. A propriedade de junção sem perdas é analisada em Aho *et al.* (1979). Os algoritmos para determinar as chaves de uma relação a partir das dependências funcionais são dados em Osborn (1976); o teste para a BCNF é discutido em Osborn (1979). O teste para a 3FN é discutido em Tsou e Fischer (1982). Os algoritmos para o projeto de relações na BCNF são dados em Wang (1990) e Hernandez e Chan (1991).

As dependências multivaloradas e a quarta forma normal são definidas em Zaniolo (1976) e Nicolas (1978). Muitas das formas normais avançadas se devem a Fagin: a quarta forma normal em Fagin (1977), PJNF em Fagin (1979), e DKNF em Fagin (1981). O conjunto de regras sólido e completo para as dependências funcionais e multivaloradas foi dado em Beeri *et al.* (1977). As dependências de junção são discutidas em Rissanen (1977) e Aho *et al.* (1979). As regras de inferência para as dependências de junção são dadas em Sciore (1982). As dependências de inclusão são discutidas em Casanova *et al.* (1981) e analisadas mais profundamente em Cosmadakis *et al.* (1990). Sua utilização na otimização de esquemas relacionais é discutida em Casanova *et al.* (1989). Dependências *template* (de molde) são discutidas em Sadri e Ullman (1982). Outras dependências são discutidas em Nicolas (1978), Furtado (1978) e Mendelzone Maier (1979). Abiteboul *et al.* (1995) fornecem um tratamento teórico para muitas das idéias apresentadas neste capítulo e no Capítulo 10.

12

Metodologia para Projeto Prático de Banco de Dados e Uso de Diagramas UML

Neste capítulo transitaremos da teoria à prática em projetos de bancos de dados. Descrevemos, em vários capítulos, o que é pertinente ao projeto de bancos de dados para aplicações práticas no mundo real. Nesse sentido incluem-se os capítulos 3 e 4 para modelagem conceitual de banco de dados; os capítulos 5 a 9 para o modelo relacional, linguagem SQL, álgebra e cálculo relacional, mapeamento de esquemas de alto-nível conceitual ER ou EER em um esquema relacional, e programação em sistemas relacionais (SGBDRs); e os capítulos 10 e 11 para teoria de dependência de dados e algoritmos para normalização relacional.

O projeto de um banco de dados global deve possuir um processo sistemático chamado **metodologia de projeto**, caso o banco de dados projetado seja administrado por um SGBDR, ou por um sistema gerenciador orientado a objetos (SGBDO), ou por um sistema gerenciador de banco de dados objeto-relacional (SGBDOR). Várias metodologias de projeto estão implícitas nas ferramentas de projeto dos bancos de dados hoje fornecidos comercialmente. Ferramentas populares incluem Designer 2000 da Oracle; ERWin, BPWin e Paradigma Plus da Platinum Technology; Sybase Enterprise Application Studio; ER Studio de Embarcadero Technologies; System Architect da Popkin Software, entre muitos outros. Nosso objetivo, neste capítulo, não é discutir nenhuma metodologia específica, mas o projeto de banco de dados em um contexto mais amplo, como é tratado em grandes organizações, para projeto e implementação de aplicações que suprem centenas ou milhares de usuários.

Geralmente, o projeto de pequenos bancos de dados, com até cerca de 20 usuários, não é complicado. Mas para bancos de dados de médio ou grande porte, que servem a várias aplicações, cada uma com dezenas ou centenas de usuários, é necessária uma abordagem mais sistemática para a atividade de projeto do banco de dados global. O volume de dados do banco de dados não reflete a complexidade do projeto; seu esquema é que é mais importante. Qualquer banco de dados com um esquema que inclua mais de 30 ou 40 tipos entidade, e um número semelhante de tipos relacionamento, exige uma cuidadosa metodologia de projeto.

Usando o termo **grandes bancos de dados** (large database) para bancos de dados com várias dezenas de gigabytes de dados e um esquema com mais de 30 ou 40 tipos entidade diferentes, cobrimos uma ampla gama de bancos de dados de órgãos governamentais, indústrias e instituições financeiras e comerciais. Empresas do setor de serviços, inclusive bancos, hotéis, companhias aéreas, seguradoras e empresas de comunicações usam bancos de dados em suas operações 24 horas por dia, 7 dias por semana — conhecidas como operações *24 por 7*. As aplicações para esses bancos de dados são chamadas sistemas *de processamento de transações*, em vista do grande volume de transações e das taxas de processamento que são exigidas. Neste capítulo nos concentraremos no projeto para bancos de dados de médio — e grande — porte, no qual prevalece o processamento de transações.

Este capítulo possui vários objetivos. A Seção 12.1 discutirá o ciclo de vida dos sistemas de informação dentro das organizações, com ênfase particular nos sistemas de banco de dados. A Seção 12.2 destacará as fases de uma metodologia de projeto de banco de dados no contexto organizacional. A Seção 12.3 introduzirá os diagramas UML e dará detalhes sobre as notações de alguns deles, que são particularmente úteis no levantamento de requisitos, e executará o projeto conceitual e ló-

gico do banco de dados. Será apresentado um exemplo ilustrativo do projeto parcial de um banco de dados universitário. A Seção 12.4 introduzirá uma ferramenta popular de desenvolvimento de software, chamada Rational Rose, que tem os diagramas de UML como sua técnica de especificação principal. Características do Rational Rose, específicas para os requisitos de modelagem e de projeto de banco de dados, serão realçadas. A Seção 12.5 discutirá brevemente as ferramentas de projeto automatizado de banco de dados.

12.1 O PAPEL DOS SISTEMAS DE INFORMAÇÃO NAS ORGANIZAÇÕES

12.1.1 O Contexto Organizacional para Sistemas que Usam Banco de Dados

Os sistemas de banco de dados tornaram-se parte dos sistemas de informação de muitas organizações. Nos anos 60, os sistemas de informação eram dominados por sistemas de arquivo, mas, desde o princípio da década de 70, as organizações gradualmente foram aderindo aos sistemas de banco de dados. Para acomodar tais sistemas, muitas organizações criaram a função de administrador de banco de dados (DBA), OU mesmo departamentos de administração de banco de dados, para acompanhar e controlar as atividades ligadas ao ciclo de vida dos bancos de dados. A tecnologia da informação (TI) e a administração de recursos de informação (ARI) foram reconhecidas como sendo a chave para a administração bem-sucedida das grandes organizações. Vejamos algumas razões:

- Os dados são considerados um recurso da corporação, e sua administração e seu controle são vitais para o funcionamento efetivo da organização.
- Cada vez mais as funções nas organizações são automatizadas, aumentando a necessidade de manter grandes volumes de dados disponíveis atualizados instantaneamente.
- Como a complexidade dos dados e das aplicações cresce continuamente, relacionamentos complexos entre os dados precisam ser modelados e mantidos.
- Em muitas organizações há uma tendência à consolidação de recursos de informação.
- Muitas organizações estão reduzindo seus custos com pessoal, delegando ao usuário final a execução de transações empresariais. Isso se evidencia nos serviços de viagem, financeiros, vendas de bens de consumo on-line e comércio eletrônico cliente/empresa, como a Amazon.com ou a Ebay. Nesses exemplos, um banco de dados operacional, de acesso e atualização pública, deve ser projetado e estar disponível para essas transações.

Os sistemas de banco de dados satisfazem esses requisitos plenamente. Duas características adicionais desses sistemas são também muito valiosas nesses ambientes:

- *Independência de dados*, que protege os programas das mudanças subjacentes à organização lógica, nos caminhos de acesso físico e nas estruturas de armazenamento.
- *Esquemas externos* (visões), que permitem que um mesmo conjunto de dados seja usado em diversas aplicações, cada uma com sua própria visão dos dados.

Novas funcionalidades disponibilizadas pelos sistemas de banco de dados e as características-chave que eles oferecem os fazem componentes integrantes dos sistemas de informação baseados em computador:

- Integração dos dados, em várias aplicações, por meio de um único banco de dados.
- Simplicidade no desenvolvimento de novas aplicações que usam linguagens de alto-nível, como SQL.
- Possibilidade de acessos eventuais por *browsing* (navegação) e de consultas para gerentes e apoio à maior parte do processamento de transações em nível de produção.

Do início dos anos 70 até meados da década de 80, a tendência era a criação de grandes repositórios centralizados de dados, administrados por um único SGBD centralizado. Durante os últimos 10 a 15 anos, essa tendência foi invertida por conta do desenvolvimento de:

1. Computadores pessoais e sistemas de banco de dados — produtos de software como EXCEL, FOXPRO, ACCESS (todos da Microsoft) ou SQL Anywhere (da Sybase), e produtos de domínio público como MYSQL — estão sendo utilizados largamente por usuários que, anteriormente, pertenciam à categoria de usuários casuais e ocasionais. Muitos administradores, secretárias, engenheiros, cientistas, arquitetos etc. pertencem a esta categoria. Como resultado, a prática de criar bancos de dados pessoais está ganhando popularidade. É possível, atualmente, copiar parte de um

grande banco de dados, de um computador *mainframe* ou de um servidor de banco de dados, e trabalhar com ele em uma estação de trabalho pessoal, para depois recolocá-lo novamente no *mainframe*. De maneira análoga, os usuários podem projetar e criar seus próprios bancos de dados para depois fundi-los em um banco maior.

2. O advento da distribuição e dos SGBDs cliente-servidor (Capítulo 25) cria a opção de distribuir o banco de dados a diversos sistemas de computador, para maior controle local e um processamento local mais rápido. Ao mesmo tempo, usuários locais podem acessar dados remotos usando as facilidades fornecidas pelo SGBD, como um cliente ou por meio da Web. Ferramentas de desenvolvimento de aplicação, como PowerBuilder ou Developer 2000 (da Oracle), estão sendo usadas largamente, como facilidade embutida, para ligar aplicações suportadas por diferentes servidores de banco de dados.
3. Muitas organizações usam **sistemas de dicionário de dados** ou **repositórios de informação**, mini SGBDs que administram **metadados** — isto é, dados que descrevem a estrutura dos dados, suas restrições, suas aplicações, suas autorizações, e assim por diante. Eles são usados freqüentemente como ferramenta integrante para administração dos recursos de informação. Um sistema de dicionário de dados útil deve armazenar e administrar os seguintes tipos de informação:
 - a. Descrição dos esquemas do sistema de banco de dados.
 - b. Informação detalhada do projeto de banco de dados físico, como estruturas de armazenamento, caminhos de acesso e tamanhos de arquivo e registro.
 - c. Descrições dos usuários do banco de dados, suas responsabilidades e autoridade de acesso.
 - d. Descrições em alto nível das transações do banco de dados, das aplicações e dos relacionamentos entre usuários e transações.
 - e. Relacionamento entre as transações do banco de dados e os itens de dados referenciados por elas. É **útil** para determinar quais transações são afetadas quando definições de dados forem alteradas.
 - f. Estatísticas de uso, como freqüências de consultas e transações, bem como volume de acesso a uma dada porção de dados.

Esses metadados estão disponíveis aos DBAs, aos projetistas e aos usuários autorizados, como os sistemas on-line de documentação. Isso melhora o controle dos DBAs sobre os sistemas de informação e a compreensão dos usuários no uso do sistema. O advento da tecnologia de *data warehousing* evidenciou a importância dos metadados.

O desempenho torna-se fundamental quando se projetam **sistemas de processamento de transações** de alto desempenho, que exigem operação ininterrupta. Esses bancos de dados são acessados freqüentemente por centenas de transações por minuto, de terminais locais e remotos. O desempenho das transações, em termos do número médio de transações por minuto e do tempo de resposta médio e máximo, torna-se fundamental. É imperativo, então, um projeto cuidadoso de banco de dados físico, que conheça as necessidades do processamento de transações da organização em tais sistemas.

Algumas organizações têm delegado a administração de seus recursos de informação a um determinado SGBD e a produtos de dicionário de dados. O investimento em projetos e implementações de sistemas grandes e complexos torna difícil a troca desses produtos por SGBDs mais novos, de modo que a organização acaba atrelada a seu SGBD atual. Em relação a bancos de dados grandes e complexos, não podemos deixar de enfatizar a importância de um projeto cuidadoso, que considere as possíveis necessidades de alterações no sistema — chamado sintonização (*tuning*) — para responder às mudanças necessárias de requisitos. Discutiremos sintonia com otimização de consultas no Capítulo 16. O custo pode ser muito alto se um sistema grande e complexo não puder evoluir e tornar-se necessária a migração para outros SGBDs.

12.1.2 O Ciclo de Vida dos Sistemas de Informação

Em uma grande organização, geralmente o sistema de banco de dados faz parte do **sistema de informação**, que inclui todos os recursos que são envolvidos no conjunto, bem como administração, uso e disseminação dos recursos de informação. Em um ambiente computadorizado, esses recursos incluem os próprios dados, o software do SGBD, o hardware do sistema e da mídia de armazenamento, o pessoal que usa e administra os dados (DBA, usuários finais, usuários parametrizáveis, e assim por diante), o software de aplicações que acessa e atualiza os dados, além dos programadores de aplicação, que desenvolvem essas aplicações. Assim, o sistema de banco de dados faz parte de um sistema de informação organizacional muito maior.

Nesta seção examinaremos o ciclo de vida típico de um sistema de informação e como o sistema de banco de dados se ajusta a ele. O ciclo de vida do sistema de informação é chamado freqüentemente de **ciclo de vida macro**, enquanto o ciclo de vida do banco de dados é chamado **ciclo de vida micro**. A distinção entre eles está se tornando confusa nos sistemas de infor-

mação, nos quais os bancos de dados são os principais componentes integrantes. O ciclo de vida macro normalmente inclui as seguintes fases:

1. *Análise de viabilidade*: Esta fase está relacionada à análise das áreas potenciais de aplicação, uma vez que ela identifica a economia relativa ao ganho e a disseminação da informação, executando estudos preliminares de custo-benefício, determinando a complexidade de dados e processos e estabelecendo prioridades entre as aplicações.
2. *Levantamento e análise de requisitos*: São coletados requisitos detalhados, por meio da interação com usuários potenciais e grupos de usuários, para identificação de seus problemas e de suas necessidades. São identificadas as dependências de interaplicações, a comunicação e os procedimentos para relatórios.
3. *Projeto*: Esta fase tem dois aspectos — o projeto do sistema de banco de dados e o projeto dos sistemas de aplicação (programas)—, que usam e processam o banco de dados.
4. *Implementação*: O sistema de informação é implementado, o banco de dados está carregado e as transações de banco de dados são implantadas e testadas.
5. *Validação e teste de aceitação*: A aceitabilidade do sistema é comparada aos requisitos dos usuários, e os critérios de desempenho são validados. O sistema é testado contra critérios de desempenho e especificações de comportamento.
6. *Implantação, operação e manutenção*: Esta fase pode ser precedida pela adaptação dos usuários de um sistema mais antigo ou pelo treinamento dos usuários. A fase operacional começa quando todas as funções do sistema estiverem operacionalizadas e validadas. Como sempre, surgem novas aplicações ou requisitos, que passarão por todas as fases anteriores até que sejam validados e incorporados ao sistema. A monitoração do desempenho do sistema e sua manutenção são atividades importantes durante a fase operacional.

12.1.3 Ciclo de Vida de Sistemas de Aplicações de Banco de Dados

Atividades relacionadas ao ciclo de vida de sistemas (micro) de aplicações de banco de dados incluem:

1. *Definição de sistemas*: Aqui são definidos o escopo do sistema de banco de dados, seus usuários e suas aplicações. São identificadas as interfaces para as várias categorias de usuários, as restrições para tempos de resposta e as necessidades de armazenamento e processamento.
2. *Projeto de banco de dados*: Ao término desta fase, o projeto lógico e físico completo do sistema de banco de dados, no SGBD escolhido, estará pronto.
3. *Implementação do banco de dados*: Inclui a especificação das definições conceituais, externa e interna do banco de dados, criando o banco de dados com arquivos vazios e implementando as aplicações de software.
4. *Carregamento ou conversão do banco de dados*: O banco de dados é carregado por meio da inserção direta dos dados ou pela conversão, para o formato do sistema, de arquivos existentes.
5. *Conversão das aplicações de software*: Qualquer software aplicativo do sistema anterior é convertido para o novo sistema.
6. *Teste e validação*: O sistema novo é testado e validado.
7. *Operação*: O sistema de banco de dados e suas aplicações são colocados em operação. Normalmente, o novo sistema e o antigo operam paralelamente durante algum tempo.
8. *Monitoramento e manutenção*: Durante a fase operacional, o sistema é constantemente mantido e monitorado. Pode haver crescimento e expansão de conteúdo de dados e aplicações de software. De vez em quando, podem ser necessárias modificações maiores e reorganizações.

As atividades 2, 3 e 4, juntas, fazem parte das fases de projeto e implementação da maioria dos ciclos de vida dos sistemas de informação. Nossa ênfase, na Seção 12.2, é dada às atividades 2 e 3, que cobrem as fases de projeto e de implementação do banco de dados. A maioria dos bancos de dados das organizações passa pelas atividades do ciclo de vida precedente. As atividades de conversão (4 e 5) não são aplicáveis a banco de dados e a aplicações novas. Quando uma organização passa de um sistema estabelecido para um novo, as atividades 4 e 5 normalmente são mais demoradas, e o esforço para realizá-las é freqüentemente subestimado. Em geral, há freqüentes avaliações entre os diversos passos, pois surgem novos requisitos em todas as fases. A Figura 12.1 mostra a reavaliação, que afeta as fases de projetos conceituais e lógicas, como resultado da implementação e sintonização do sistema.

12.2 O PROCESSO DE PROJETO E IMPLEMENTAÇÃO DE BANCO DE DADOS

Focalizaremos agora as atividades 2 e 3 do ciclo de vida dos sistemas de aplicação de banco de dados, que são o projeto e a implementação. O problema do projeto de banco de dados pode ser expresso conforme segue:

PROJETO LÓGICO E FÍSICO DA ESTRUTURA DE UM OU MAIS BANCOS DE DADOS PARA ACOMODAR AS NECESSIDADES DE INFORMAÇÃO DOS USUÁRIOS EM UMA ORGANIZAÇÃO PARA UM CONJUNTO DEFINIDO DE APLICAÇÕES.

São várias as metas do projeto de banco de dados:

- Satisfazer os requisitos de informações especificadas por usuários e aplicações.
- Proporcionar uma estruturação natural e fácil para entender a informação.
- Dar suporte a quaisquer requisitos de processo e objetivos de desempenho, como tempo de resposta, tempo de processamento e espaço de armazenamento.

Essas metas são muito difíceis de executar e medir, pois envolvem um intercâmbio intrínseco: se o que se busca é mais 'naturalidade' e 'entendimento' do modelo, pode-se penalizar o desempenho. O problema é agravado porque o processo de projeto do banco de dados começa, freqüentemente, com requisitos informais e mal definidos. Em contraste, o resultado da atividade do projeto é um esquema de banco de dados definido de forma rígida, que, uma vez implementado o banco de dados, não pode ser modificado facilmente. Podemos identificar seis fases principais do projeto global do banco de dados e do processo de implementação:

1. Levantamento e análise de requisitos.
2. Projeto conceitual do banco de dados.
3. Escolha de um SGBD.
4. Mapeamento do modelo de dados (também chamado projeto lógico de banco de dados).
5. Projeto físico do banco de dados.
6. Implementação e sintonização (tuning) do sistema de banco de dados.

O processo de projeto consiste em duas atividades paralelas, como ilustrado na Figura 12.1. A primeira atividade envolve o projeto do **conteúdo e da estrutura dos dados** do banco de dados; a segunda relaciona o **projeto de aplicações** ao banco de dados. Para simplificar a figura, evitamos a exibição da maioria das interações entre esses dois lados, mas as duas atividades são entrelaçadas. Por exemplo, analisando as aplicações, podemos identificar os tipos de dados que serão armazenados no banco de dados. Além disso, é durante a fase do projeto físico do banco de dados que escolhemos a estrutura de armazenamento e o caminho de acesso aos arquivos, dependendo das aplicações que usarão esses arquivos. Porém, normalmente especificamos o projeto das aplicações do banco de dados recorrendo ao esquema construído para ele, definido durante a primeira atividade. Claramente, essas duas atividades têm forte influência uma sobre a outra. Tradicionalmente, as metodologias de projeto de banco de dados enfatizam a primeira dessas atividades, considerando que o projeto de software focalizaria a segunda; este pode ser chamado **projeto dirigido-pelos-dados** (*data-driven*) versus **projeto dirigido-pelo-processo** (*process-driven*). Os projetistas de banco de dados e os engenheiros de software cada vez mais vêm reconhecendo que as duas atividades deveriam se desenvolver conjuntamente, e as ferramentas de projeto cada vez mais vêm combinando as duas.

As seis fases mencionadas não precisam seguir estritamente a seqüência descrita. Em muitos casos, temos de projetar uma fase antes da outra. Esses *feedbacks* entre as fases — e também dentro de cada fase — são comuns. Mostramos somente um par de *feedbacks* na Figura 12.1, mas há muitos outros entre os vários pares de fases. Também mostramos algumas interações entre dados e processos na figura; na realidade, há muitas outras interações também. A Fase 1, da Figura 12.1, envolve o levantamento dos requisitos planejados para o banco de dados, e as preocupações da Fase 6 recaem na implementação e no reprojeto do banco de dados. O coração do processo de projeto de banco de dados engloba as Fases 2, 4 e 5; resumiremos essas fases sucintamente:

- Projeto *conceitual do banco de dados* (Fase 2): A meta desta fase é produzir um esquema conceitual do banco de dados, que é independente de um SGBD específico. Durante esta fase, usamos freqüentemente um modelo de dados de alto nível, como os modelos ER ou EER (capítulos 3 e 4). Além disso, tanto quanto possível, especificaremos as aplicações ou transações conhecidas do banco de dados, usando uma notação independente de qualquer SGBD específico. Frequentemente, a escolha do SGBD é feita pela organização; a intenção do projeto conceitual é manter-se tão independente quanto possível das considerações de implementação.

Mapeamento do modelo de dados (Fase 4): Durante esta fase, que também é chamada de **projeto lógico do banco de dados**, **mapeamos** (ou **transformamos**) o esquema de dados do modelo conceitual de alto nível usado na Fase 2 para dados no modelo do SGBD escolhido. Podemos começar esta fase depois de escolher um tipo específico de SGBD por exemplo, se decidirmos usar algum SGBD relacional, mas ainda não tivermos decidido qual em particular. Chegamos o resultado de projeto lógico de *sistema-independente* (embora *dependente do modelo de dados*). Em termos de arquitetura de três níveis de SGBD, discutida no Capítulo 2, o resultado desta fase é o *esquema conceitual no modelo de dados* escolhido. Além disso, o projeto dos *esquemas externos* (visões) para aplicações específicas normalmente é realizado durante esta fase.

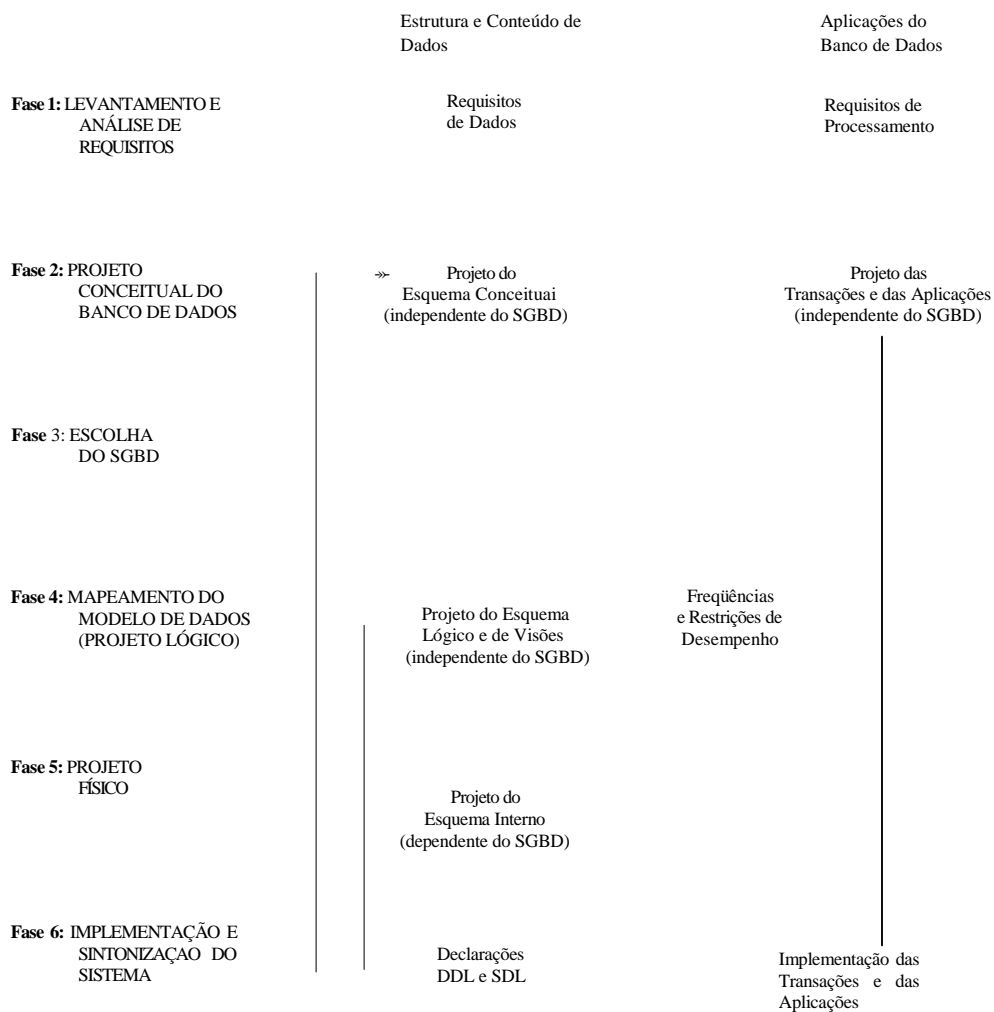


FIGURA 12.1 Fases do projeto e da implementação para grandes bancos de dados.

Projeto físico do banco de dados (Fase 5): Durante esta fase, projetamos as especificações para o banco de dados armazenado em termos do armazenamento físico das estruturas, da alocação de registros e dos índices. Corresponde projetar o esquema interno na terminologia da arquitetura de três níveis do SGBD.

Implementação e sintonização do sistema de banco de dados (Fase 6): Nesta fase, o banco de dados e os programas de aplicação são implementados, testados e eventualmente utilizados em produção. São testadas também várias transações e aplicações individualmente e, depois, em conjunto. Normalmente, esta fase revela necessidades de alterações no projeto físico, na indexação, na reorganização e na alocação de dados — atividade que chamamos de **sintoniza** (afinação, *tuning*) do **banco de dados**. Sintonizar é uma atividade contínua — parte da manutenção do sistema que perdura durante todo o ciclo de vida de um banco de dados, contanto que o banco de dados e as aplicações continuem evoluindo ou à medida que forem surgindo problemas de desempenho.

Nas subseções seguintes, discutiremos com mais detalhes cada uma das seis fases de projeto de banco de dados.

12.2.1 Fase 1: Levantamento e Análise de Requisitos¹

Antes de projetarmos efetivamente um banco de dados, temos de conhecer e analisar as expectativas dos usuários e os usos planejados com o máximo de detalhes possíveis. Esse processo é chamado **levantamento e análise de requisitos**. Para especificar os requisitos, devemos primeiro identificar todas as partes do sistema de informação que interagirão com o sistema. Essas partes incluem usuários, novos e existentes, e aplicações, cujos requisitos são então coletados e analisados. Geralmente fazem parte desta fase:

1. Identificação das principais áreas de aplicação, bem como dos grupos de usuários do banco de dados, ou seja, quem terá seu trabalho afetado por ele. São escolhidas as pessoas-chave e os comitês dentro de cada grupo para apoiar os passos subsequentes de coleta e especificação dos requisitos.
2. Análise e estudo da documentação existente relativa às aplicações. Outros documentos — manuais de procedimentos, formulários, relatórios e organogramas — são revisados para determinar se têm qualquer influência no levantamento de requisitos e no processo de especificação.
3. O estudo do ambiente operacional atual e o uso planejado da informação, incluindo a análise dos tipos de transação e sua frequência, bem como o fluxo de informação dentro do sistema. São estudadas as características geográficas relativas aos usuários, origem das transações, destino de relatórios e assim por diante. São especificadas as entradas e as saídas das transações.
- 4- Respostas a conjuntos de consultas levantadas por usuários ou grupos de usuários potenciais do banco de dados. Essas consultas envolvem as prioridades dos usuários e a importância que eles atribuem às diversas aplicações. Pessoas-chave podem ser entrevistadas para ajudar a definir o valor da informação e o estabelecimento de prioridades.

A análise de requisitos é levada a cabo para os usuários finais do sistema de banco de dados, ou 'clientes', por um time de analistas ou peritos em requisitos. Provavelmente os requisitos iniciais serão informais, incompletos, incompatíveis e apenas parcialmente corretos. Muito trabalho terá de ser feito, então, para transformar esses requisitos iniciais em uma especificação de aplicação que possa ser usada pelos implementadores e pelos avaliadores como ponto de partida para a implementação e para os casos de teste. Os requisitos refletem o entendimento inicial de um sistema que ainda não existe, e eles inevitavelmente mudarão. Portanto, é importante usar técnicas que ajudem os clientes a convergir rapidamente para os requisitos da implementação.

A participação do cliente no processo de desenvolvimento aumenta a satisfação com o sistema a ser entregue. Por isso, muitos profissionais atualmente fazem reuniões e workshops para envolver todos os 'donos' do projeto. Uma dessas metodologias para o refinamento inicial dos requisitos do sistema é chamada Projeto de Aplicação Conjunto (JAD — *Join Application De-sign*). Mais recentemente foram desenvolvidas técnicas, como o Projeto Contextual, que propõem introduzir os projetistas no ambiente de trabalho no qual será utilizada a aplicação. Para ajudar os representantes dos clientes a entender melhor o sistema proposto, é comum fazê-los transitar pelo fluxo de trabalho ou pelos cenários das transações, ou criar um protótipo da aplicação.

Os procedimentos descritos ajudam a definir a estrutura e refinam os requisitos, mas ainda nos deixam em um estado informal. Para transformar esses requisitos em uma forma mais estruturada, são usadas **técnicas para especificação de requisitos**, que incluem OOA (análise orientada a objeto), DFDs (diagramas de fluxos de dados) e refinamento dos objetivos da aplicação. Esses métodos utilizam técnicas de diagramação para organizar e apresentar os requisitos de informação-processo. Documentação adicional, na forma de textos, tabelas, quadros e requisitos de decisão normalmente acompanham os diagramas. Há técnicas que produzem uma especificação formal que pode ser checada matematicamente pela consistência e por análises simbólicas 'qual-se'. Esses métodos dificilmente são utilizados hoje, mas podem tornar-se padrão no futuro para parte dos sistemas de informação que contemplam funções de missão-crítica e que, portanto, precisam funcionar como planejado. Os procedimentos para especificação formal, modelo-baseados, dos quais a metodologia e a notação-Z é a mais proeminente, podem ser pensados como extensões do modelo ER e são, portanto, os mais aplicáveis para projetos de sistemas de informação.

Algumas técnicas apoiadas por computador (*computer-aided*) — chamadas ferramentas 'Upper CASE' — foram propostas para ajudar a checar a consistência e a 'completeza' das especificações, que normalmente são armazenadas em um único repositório e podem ser exibidas e atualizadas durante o progresso do projeto. Outras ferramentas são usadas para rastrear a relação entre os requisitos e as outras entidades do projeto, como os módulos de código e os casos de teste. Esses *bancos de dados de rastreamento* são especialmente importantes em conjunto com os procedimentos de gestão de mudança para sistemas nos quais são frequentes as alterações de requisitos. Eles também são usados em projetos contratuais, nos quais a empresa respon-

Diagramas UML

sável pelo desenvolvimento deve proporcionar evidência documental ao cliente, provando que todos os requisitos foram implementados.

As fases de levantamento de requisitos e de análise podem ser bastante demoradas, mas são cruciais para o sucesso do sistema de informação. Corrigir um erro de definição de requisito é muito mais caro que corrigir um erro de implementação pois os efeitos de um erro de requisito são normalmente graves e muito mais trabalhosos para serem reimplementados. Não corrigir o erro significa que o sistema não irá satisfazer o cliente e, quem sabe, nem mesmo será utilizado. Levantamento e análise de requisitos têm sido assunto de livros inteiros.

12.2.2 Fase 2: Projeto Conceitual do Banco de Dados

A segunda fase do projeto de banco de dados envolve duas atividades paralelas. A primeira, o projeto do esquema conceitual, examina os requisitos de dados, que são o resultado da Fase 1, e produz um esquema conceitual do banco de dados. A segunda, o projeto das transações e das aplicações, examina as aplicações do banco de dados, analisadas na Fase 1, e produz especificações de alto nível para essas aplicações.

Fase 2a: Projeto do Esquema Conceitual. O esquema conceitual produzido nesta fase normalmente está contido em um modelo de dados de alto nível, independentemente do SGBD, pelas seguintes razões:

1. O objetivo do projeto do esquema conceitual é a completa compreensão da estrutura do banco de dados, do significado (semântica), dos inter-relacionamentos e das restrições. Esse objetivo é mais facilmente alcançado se for independente de um SGBD específico, porque cada SGBD tem idiosincrasias e restrições que normalmente não deveriam influenciar o projeto do esquema conceitual.
2. O esquema conceitual é inestimável como *descrição estável* do conteúdo do banco de dados. A escolha do SGBD, bem como as posteriores decisões de projeto, podem ser alteradas sem que haja necessidade de modificar o esquema conceitual SGBD-independente.
3. O bom entendimento do esquema conceitual é crucial para os usuários do banco de dados e para os projetistas das aplicações. O uso de um modelo de dados de alto nível, mais expressivo e genérico que o modelo de dados de um SGBD em particular, é conseqüentemente muito importante.
4. A descrição diagramática do esquema conceitual serve como excelente veículo de comunicação entre usuários do banco de dados, projetistas e analistas. Os modelos de dados de alto nível normalmente apresentam conceitos que são mais facilmente compreendidos que os modelos de baixo nível dos SGBDs, ou as definições sintáticas dos dados; pois qualquer comunicação relativa ao esquema do projeto torna-se mais exata e mais direta.

Nesta fase de projeto de banco de dados é importante usar dados conceituais de alto nível com as seguintes características:

1. *Expressividade*: O modelo de dados deve ser expressivo o bastante para distinguir tipos diferentes de dados, relacionamentos e restrições.
2. *Simplicidade e inteligibilidade*: O modelo deve ser simples o bastante para que usuários não-especializados possam entender e usar seus conceitos.
3. *Sintético*: O modelo deve ter um número reduzido de conceitos básicos que sejam distintos e sem sobreposição de significados.
4. *Representação diagramática*: O modelo deve ter uma notação diagramática para exibir o esquema conceitual de modo que seja de fácil interpretação.
5. *Formalismo*: Um esquema conceitual expresso em um modelo de dados deve representar uma especificação formal não ambígua dos dados. Conseqüentemente, os modelos conceituais devem ser definidos com precisão e sem ambigüidade.

Muitos desses requisitos — o primeiro em particular — às vezes entram em conflito com os demais. Muitos modelos conceituais de alto nível foram propostos para o projeto de banco de dados (bibliografia selecionada no Capítulo 4). Na discussão seguinte, usaremos a terminologia do modelo Entidade-Relacionamento Estendido (EER), apresentado no Capítulo 4 e mostraremos que ele estará sendo usado nesta fase. O projeto do esquema conceitual, inclusive a modelagem de dados, está se tornando parte integrante da análise orientada a objeto e das metodologias de projeto. A UML tem diagramas de classe que estão, em grande parte, baseados em extensões do modelo EER.

- 2 Essa fase do projeto é discutida com detalhes nos primeiros sete capítulos de Batini *et al.* (1992); resumiremos aqui essa discussão.

Abordagens para o Projeto do Esquema Conceitual. Para o projeto de um esquema conceitual, temos de identificar os componentes básicos do esquema: tipos entidade, tipos relacionamento e atributos. Deveríamos também especificar atributos fundamentais, cardinalidades e restrições de participação nos relacionamentos, nos tipos entidade fraca e nas hierarquias/reticulados de especialização/generalização. Há duas abordagens de projeto para o esquema conceitual que são derivadas dos requisitos levantados durante a Fase 1.

A primeira abordagem é a do **projeto centralizado do esquema** (ou **um-tiro-só**), na qual os requisitos da Fase 1, antes mesmo do início do projeto do esquema, originados a partir de diversas aplicações e usuários, são agrupados em um só conjunto de requisitos. Um único esquema, que corresponde ao conjunto unificado dos requisitos, é, então, projetado. Quando existem muitos usuários e aplicações, a fusão de todos os requisitos pode ser uma tarefa árdua e demorada. Supõe-se que uma autoridade centralizadora, o DBA, seja responsável por decidir como fundir os requisitos e como projetar o esquema conceitual para o banco de dados inteiro. Uma vez projetado e finalizado o esquema conceitual, os esquemas externos dos diversos grupos de usuários e aplicações podem ser especificados pelo DBA.

A segunda abordagem é a da **integração** de visões, na qual os requisitos não são fundidos. Quando um esquema (ou visão) é projetado para um grupo de usuários ou para aplicação, ele apenas se baseia em seus próprios requisitos. Assim, desenvolvemos um esquema de alto nível (visão) para cada grupo de usuários ou aplicação. Numa fase posterior de **integração de visões**, esses esquemas são fundidos ou integrados em um único esquema **conceitual global** para o banco de dados como um todo. As visões individuais podem ser reconstruídas como esquemas externos, depois da integração das visões.

A principal diferença entre as duas abordagens é a maneira e o estágio em que se organizará a fusão das diversas visões ou os requisitos dos vários usuários e aplicações. Na abordagem centralizada, a fusão é feita manualmente pelo grupo de DBAs antes de projetar qualquer esquema, e será aplicada diretamente aos requisitos levantados na Fase 1. Dessa forma, coloca-se o peso da integração das diferenças e dos conflitos entre os grupos de usuários no grupo de DBAs. O problema é normalmente negociado por meio da contratação de consultores/projetistas externos, que trazem suas soluções para tais conflitos. Por conta das dificuldades em administrar essa tarefa, a abordagem de integração de visões vem ganhando maior aceitação.

Na abordagem de integração de visões, cada grupo de usuário ou aplicação projeta o esquema conceitual (EER) de seus próprios requisitos. Então um processo de integração é aplicado a esses esquemas (visões) pelo DBA, a fim de formar o esquema integrado global. Embora a integração de visões possa ser feita manualmente, um grande banco de dados, que envolve dezenas de grupos de usuários, exige metodologia e uso de ferramentas automatizadas para auxiliar essa integração. A correspondência entre atributos, tipos entidade e tipos relacionamento das várias visões precisa ser especificada para que a integração possa ser realizada. Além disso, a integração de visões contraditórias e a verificação de consistência das correspondências interestemas são problemas que precisam ser negociados.

Estratégias para Projeto de Esquema. Dado um conjunto de requisitos, seja para um único usuário ou para uma grande comunidade de usuários, temos de criar um esquema conceitual que satisfaça esses requisitos. Há várias estratégias para o projeto de tal esquema. A maioria segue uma abordagem incremental — isto é, começam com algum esquema construído, derivado dos requisitos e, a partir dele, incrementam-se modificações, refinamentos ou novas construções. Discutiremos algumas dessas estratégias:

1. **Estratégia top-down (de cima para baixo):** Começamos com um esquema que contém abstrações de alto nível e, então, aplicamos sucessivos refinamentos *top-down*. Por exemplo, podemos especificar alguns poucos tipos entidade de alto nível e, então, conforme especificamos seus atributos, dividi-los em tipos entidade e relacionamento de níveis mais baixos. O processo de especialização para refinar um tipo entidade em subclasses, que ilustramos nas seções 4.2 e 4.3 (figuras 4.1, 4.4 e 4.5), é outro exemplo de estratégia de projeto *top-down*.
2. **Estratégia bottom-up (de baixo para cima):** Começamos com um esquema que contém abstrações básicas e, então, combinamos ou acrescentamos abstrações. Por exemplo, podemos começar com os atributos e depois agrupá-los em tipos entidade e relacionamento. No decorrer do projeto, podemos adicionar relacionamentos novos entre os tipos entidade. O processo de generalização de tipos entidade em superclasses generalizadas de mais alto nível (seções 4-2 e 4-3, Figura 4-3) é outro exemplo de estratégia *bottom-up* de projeto.
3. **Estratégia inside-out (de dentro para fora):** Este é um caso especial da estratégia *bottom-up*, em que é focado um conjunto central de conceitos que são muito evidentes. A modelagem, então, amplia tais conceitos, considerando os novos ao lado dos existentes. Poderíamos especificar no esquema alguns poucos tipos entidade evidentes e continuar adicionando outros tipos entidade e relacionamento relacionados a cada um.
- 4- **Estratégia mista:** Em vez de seguir uma estratégia particular ao longo de todo o projeto, os requisitos são divididos de acordo com uma estratégia *top-down*, e um esquema parcial é projetado para cada parte, de acordo com uma estratégia *bottom-up*. As várias partes do esquema são, então, combinadas.

Capítulo 12 Metodologia para Projeto Prático de Banco de Dados e Uso de Diagramas UML

As figuras 12.2 e 12.3 ilustram os refinamentos *top-down* e *bottom-up*, respectivamente. Um exemplo da primitiva de refinamento *top-down* é a decomposição de um tipo entidade em vários tipos entidade. A Figura 12.2a mostra CURSO sendo refinado em CURSO e SEMINÁRIO, e o relacionamento ENSINA dividido em ENSINA e OFERECE. A Figura 12.2b mostra um tipo entidade CURSO_OFERECIDO refinado em dois tipos entidade (CURSO e INSTRUTOR), com um relacionamento entre eles. O refinamento normalmente força o projetista a ter um maior questionamento e a extrair mais restrições e detalhes: por exemplo, o (min, max) e cardinalidade são obtidos entre o relacionamento CURSO e INSTRUTOR durante o refinamento. A Figura 12.3a mostra a primitiva de refinamento *bottom-up* gerando relacionamentos novos entre tipos entidade. O refinamento *bottom-up*, que usa categorização (tipo união, agregado), é ilustrado na Figura 12.3b, na qual o conceito novo de VEICULO_PROPRIETARIO é descoberto a partir dos tipos entidade existentes: DOCENTE, FUNCIONÁRIO e ALUNO. Esse processo de criar uma categoria e a notação diagramática relacionada corresponde ao que foi apresentado na Seção 4-4.

(a)

DOCENTE

d,N)

ENSINA

4

(1,3)

CURSO

(1,N) ,< ENSINA

DOCENTE

(1,1)

CURSO

(1,3)

SEMINÁRIO

CURSO

(1,1) ^ "\ (1,N)

OFERECIDO_POR— INSTRUTOR

FIGURA 12.2 Exemplos de refinamento *top-down*. (a) Gerando um tipo entidade novo. (b) Decompondo um tipo entidade em dois tipos: entidade e relacionamento.

Integração de Esquemas (*de Visões — Views*). Para grandes bancos de dados, com muitos usuários e aplicações, pode ser usada a abordagem de integração de esquemas de visões individuais, mescladas posteriormente. As visões individuais são relativamente pequenas, e o projeto dos esquemas é simplificado. Porém, será necessária uma metodologia para integrar essas visões a um esquema de banco de dados global. A integração de esquemas pode ser dividida nas seguintes subtarefas:

1. *Identificando correspondências e conflitos entre os esquemas*: Como os esquemas são projetados individualmente, é necessário especificar construtores que representem o mesmo conceito do mundo real. Tais correspondências deverão ser identificadas antes de efetuar a integração. Durante esse processo, podem ser descobertos vários tipos de conflitos entre os esquemas:

a. *Conflitos de nome*: Podem ser de dois tipos — sinônimos e homônimos. Um sinônimo ocorre quando dois esquemas usam nomes diferentes para descrever o mesmo conceito; por exemplo, o tipo entidade USUÁRIO de um esquema pode descrever o mesmo conceito do tipo entidade CLIENTE de outro esquema. Um homônimo ocorre quando dois esquemas usam o mesmo nome para descrever conceitos diferentes; por exemplo, um tipo entidade PEÇA pode representar componentes de um computador em um esquema e peças de mobiliário em outro.

12.2 O Processo de Projeto e Implementação de Banco de Dados 267

(a)
DOCENTE
ALUNO
(b)
VAGA_ESTACIONAMENTO
VAGA,,ESTACIONAMENTO
DOCENTE
ADMINISTRATIVO
PROPRIETARIO_VEICULO
ALUNO

n

DOCENTE
ADMINISTRATIVO
ALUNO

FIGURA 12.3 Exemplos de refinamento *bottom-up*. (a) Descobrindo e adicionando novos relacionamentos, (b) Descobrindo e relacionando uma nova categoria (tipo união).

- b. *Conflitos de tipo*: Um mesmo conceito pode ser representado em dois esquemas por diferentes construtores. Por exemplo, o conceito de DEPARTAMENTO pode ser um tipo entidade em um esquema e um atributo em outro.
 - c. *Conflitos de domínio (conjunto de valores)*: Um atributo pode ter domínios diferentes em dois esquemas. Por exemplo, o SSN pode ser declarado como inteiro em um esquema e como cadeia de caracteres em outro. Um conflito da unidade de medida pode ocorrer em esquemas que representam PESO em libras e em quilogramas.
 - d. *Conflitos entre restrições*: Dois esquemas podem impor restrições diferentes. Por exemplo, a chave de um tipo entidade pode ser diferente em cada esquema. Outros exemplos envolvem restrições estruturais diferentes em relacionamentos como ENSINA; um esquema pode representá-lo como 1:N (um curso tem apenas um professor), enquanto outro o representa como M:N (um curso pode ter vários professores).
2. *Modificando visões para conformar um ao outro*: Alguns esquemas são modificados de forma a se adaptarem a outros. Alguns dos conflitos identificados na primeira subetapa serão resolvidos durante este passo.
3. *Fundindo visões*: O esquema global é criado por meio da composição dos esquemas individuais. Conceitos correspondentes serão representados apenas uma vez no esquema global e são especificados no mapeamento entre as visões e o esquema global. Esse é o passo mais difícil em banco de dados do mundo real, que contém centenas de entidades e relacionamentos, pois envolve considerável intervenção humana e negociações para solucionar os conflitos e alcançar soluções razoáveis e aceitáveis ao esquema global.
- 4- *Reestruturação*: Como passo final opcional, o esquema global pode ser reavaliado e reestruturado para remover qualquer redundância ou complexidade desnecessária.
- Algumas dessas idéias são ilustradas no exemplo, bastante simplificado, apresentado nas figuras 12.4 e 12.5. Na Figura 12.4 são combinadas duas visões para criar um banco de dados para publicações. Durante a identificação de correspondências entre as duas visões, descobrimos que PESQUISADOR e AUTOR são sinônimos (até onde é pertinente para este banco de dados), como são também CONTRIBULPARA e ESCRITO_POR. Mais adiante, decidimos modificar a VISÃO 1 para incluir ASSUNTO em ARTIGO, como mostrado na Figura 12.4, conforme a VISÃO 2. A Figura 12.5 mostra o resultado da fusão da VISÃO 1 modificada pela VISÃO 2. Generalizamos o

Capítulo 12 Metodologia para Projeto Prático de Banco de Dados e Uso de Diagramas UML

tipo entidade ARTIGO e LIVRO no tipo entidade PUBLICAÇÃO com o atributo Título em comum. Os relacionamentos CONTRIBUIR PARA e ESCRITO POR foram mesclados, bem como os tipos entidade PESQUISADOR e AUTOR. O atributo Editor só se aplica ao tipo entidade LIVRO, enquanto o atributo Tamanho e o tipo relacionamento PUBLICADO EM só se aplicam a ARTIGO.

PESQUISADOR

PERIÓDICO

VISÃO 1

AUTOR

VISÃO 1 MODIFICADA

FIGURA 12.4 Modificando visões para adaptação antes da integração.

O exemplo anterior ilustra a complexidade do processo de fusão e como o significado dos vários conceitos deve ser considerado para simplificar o projeto do esquema resultante. Para projetos do mundo real, o processo de integração de esquemas requer uma abordagem mais disciplinada e sistemática. Foram propostas várias estratégias para o processo de integração de visões (Figura 12.6):

1. *integração em etapas binárias*: Os dois esquemas mais semelhantes são integrados primeiro. O esquema resultante é então integrado a outro esquema, e o processo é repetido até que todos os esquemas sejam integrados. A ordenação dos esquemas para integração pode basear-se, em alguma medida, na semelhança entre eles. Essa estratégia é satisfatória para integração manual por conta da sua abordagem passo a passo.

12.2 O Processo de Projeto e Implementação de Banco de Dados

269

2. *integração ri'ária*: Todas as visões são integradas em um único procedimento, após análise e especificação das correspondências. Em grandes projetos, esta estratégia requer ferramentas computadorizadas. Tais ferramentas foram construídas como protótipos em pesquisas, entretanto, não estão disponíveis comercialmente.

3. *Estratégia binária balanceada*: Primeiro, são integrados pares de esquemas; então, os esquemas resultantes são comparados para nova integração; o procedimento é repetido até resultar no esquema global final.

4- *Estratégia mista*: Inicialmente, os esquemas são divididos em grupos baseados na semelhança entre eles, e cada grupo é integrado separadamente. Os esquemas intermediários se agrupam novamente e são, então, integrados, e assim por diante.

LIVRO

FIGURA 12.5 Esquema integrado depois da fusão das visões 1 e 2.

Esquema integrado

Esquema

Intermediário

Integrado

Esquema integrado

Integração em Etapas Binárias**Integração N-ária**

Esquema integrado

Esquema integrado

V, V₂**Integração Binária Balanceada****Integração Mista**

FIGURA 12.6 Diferentes estratégias para o processo de integração de visões.

Fase 2b: Projeto das Transações. O propósito da Fase 2b, que ocorre em paralelo com a Fase 2a, é projetar as características das transações (aplicações) conhecidas do banco de dados de modo SGBD-independente. Quando um sistema de banco de dados é projetado, os projetistas conhecem muitas das aplicações (ou transações) que serão executadas quando ele for implementado. Uma parte importante do projeto é a especificação das características funcionais dessas transações

270 Capítulo 12 Metodologia para Projeto Prático de Banco de Dados e Uso de Diagramas UML

logo no início do processo do projeto. Isso assegura que o esquema do banco de dados incluirá todas as informações necessárias a essas transações. Além disso, conhecer a importância relativa das várias transações e de suas taxas esperadas de chamada compreende uma parte crucial do projeto físico do banco de dados (Fase 5). Normalmente, só algumas transações são conhecidas quando ainda estão em projeto, uma vez que depois que o sistema de banco de dados for implementado, novas transações serão continuamente identificadas e implementadas. Porém, as transações mais importantes são usualmente conhecidas com antecedência e deveriam ser especificadas em fases anteriores à implementação. A regra informal '80-20' normalmente se aplica a este contexto: 80% da carga de trabalho é equivalente a 20% das transações mais usadas, as quais regem o projeto. Em aplicações provenientes de consultas ad hoc ou processamento em lote (*batch*), as consultas e as aplicações que processam uma quantidade significativa de dados devem ser identificadas.

Uma técnica comum para especificar transações em nível conceitual é identificar suas **entradas/saídas** e seu **comportamento funcional**. Especificando os parâmetros de entrada e saída (argumentos) e o fluxo funcional interno de controle, os projetistas podem determinar uma transação de modo conceitual e independente do sistema. Normalmente, as transações podem ser agrupadas em três categorias: (1) transações **de recuperação**, usadas para recuperar dados para exibição em uma tela ou para produção de um relatório; (2) **transações de atualização**, usadas para entrada de dados ou para modificação de dados existentes no banco de dados; (3) **transações mistas**, usadas para aplicações mais complexas, que fazem recuperação e atualização. Por exemplo, considere um banco de dados de reservas em linhas aéreas. Uma transação de recuperação poderia listar todos os vôos matutinos de uma determinada data entre duas cidades. Uma transação de atualização poderia ser feita para reservar um assento em um vôo particular. Uma transação mista poderia exibir alguns dados, como mostrar a reserva de um cliente em algum vôo e, a seguir, atualizar o banco de dados, ou seja, cancelar a reserva ou acrescentar um trecho de vôo a uma reserva existente. As transações (aplicações) podem ser originadas em uma ferramenta *front-end* como PowerBuilder 9.0 (da Sybase) ou Developer 2000 (da Oracle), que coletam parâmetros on-line e, então, enviam uma transação ao SGBD como um *backend*:

Várias técnicas para especificação de requisitos incluem notações para especificação de **processos**, que, nesse contexto, são operações mais complexas que podem englobar diversas transações. Ferramentas para a modelagem de processos, como o BPWin, bem como ferramentas que modelam o fluxo de trabalho (*workflow*), estão se tornando populares na identificação de fluxos de informações nas organizações. A linguagem UML, que proporciona modelagem de dados por meio de diagramas de classes e objetos, tem uma variedade de diagramas para modelagem de processos, inclusive diagramas de transição de estados, diagramas de atividades, diagrama de sequência e diagramas de colaboração. Todos eles recorrem a atividades, eventos e operações dentro do sistema de informação, bem como às entradas e às saídas dos processos e ao seqüenciamento ou à sincronização dos requisitos, entre outras condições. É possível refinar essas especificações e extrair delas transações individuais. Outras propostas, como a TAXIS, a GALILEO e a GORDAS são utilizadas para a especificação de transações (bibliografia selecionada ao término deste capítulo). Algumas delas foram implementadas em protótipos de sistemas e ferramentas. A modelagem de processos ainda é uma área ativa de pesquisa.

O projeto de transações é tão importante quanto o de esquema, mas é freqüentemente considerado parte do software em vez de parte do projeto do banco de dados. Muitas das metodologias de projeto atuais enfatizam uma em detrimento da outra. Ela deveria ocorrer em paralelo com as fases 2a e 2b, com os feedbacks sendo usados para refinamento, até que seja alcançado um projeto estável de esquema e transações.

12.2.3 Fase 3: Escolha de um SGBD

A escolha de um SGBD é regida por vários fatores — alguns técnicos, outros econômicos, e outros ainda ligados à política da organização. Os fatores técnicos se preocupam com a conveniência do SGBD em relação à tarefa que se tem. Aspectos a considerar: o tipo de SGBD (relacional, objeto-relacional, objeto, outros); a estrutura de armazenamento e os caminhos de acesso que o SGBD suporta; as interfaces disponíveis para usuários e programadores; os tipos de linguagens de consulta de alto nível; a disponibilidade de ferramentas de desenvolvimento; a habilidade para conectar-se a outros SGBDs por meio de interfaces padrão; as opções arquitetônicas relacionadas a operações cliente-servidor, e assim por diante. Fatores não técnicos incluem a situação financeira e a estrutura de suporte do vendedor. Nesta seção nos concentraremos na discussão dos fatores econômicos e organizacionais que afetam a escolha do SGBD. Os seguintes custos devem ser considerados:

3 Essa filosofia foi seguida por mais de 20 anos em produtos populares, como o CICS, que funciona como uma ferramenta para gerar transações para SGBDs legados, como o IMS.

4 A modelagem de alto nível das transações é coberta por Batini *et al.* (1992, capítulos 8,9 e 11). A junção funcional e a filosofia de análise de dados são defendidas ao longo deste livro.

12.2 O Processo de Projeto e Implementação de Banco de Dados

271

1. *Custo para aquisição do software*: Este é o custo inicial mais alto para a compra de software e engloba as opções de linguagens, as diferentes interfaces, como formulários, menus e ferramentas Web para interface gráfica com o usuário (GUI), as opções de backup, os métodos de acesso especiais e a documentação. É necessário selecionar a versão correta do SGBD para um sistema operacional específico. Normalmente, as ferramentas de desenvolvimento e de projeto e suporte adicional à linguagem não estão incluídas na estimativa básica.

2. *Custo de manutenção*: Este custo ocorre periodicamente ao contratar a manutenção padrão do vendedor ou para manter a versão do SGBD atualizada.

3. *Custo de aquisição de hardware*: Pode ser necessária a aquisição de um hardware novo, como memória adicional, terminais, unidades de disco e controladoras, ou memória especializada para SGBD e memória de arquivo.

4. *Custo de criação e de conversão do banco de dados*: Este é o custo para criar o sistema de banco de dados pela primeira vez, ou para converter um sistema existente no novo software do SGBD. No último caso, é habitual operar o sistema já existente paralelamente ao novo sistema até que todas as aplicações sejam completamente implementadas e testadas. Este custo é difícil de ser estimado e, em geral, é subestimado.

5. *Custo de pessoal*: A primeira aquisição de software SGBD é freqüentemente acompanhada pela criação, ou reorganização, do departamento de processamento de dados. Há, na maioria das empresas que adotaram SGBDs, cargos para pessoal de DBA e sua equipe.

6. *Custo de treinamento*: Como os SGBDs são sistemas complexos, freqüentemente há necessidade de treinamento de pessoal para usar e programar o SGBD. O treinamento é necessário em todos os níveis, inclusive em programação, desenvolvimento de aplicações e administração de banco de dados.

7. *Custo operacional*: O custo da operação continuada do sistema de banco de dados não é normalmente considerado nas alternativas de avaliação porque ele ocorre independentemente do SGBD selecionado.

Os benefícios advindos da aquisição de um SGBD não são fáceis de medir e quantificar, pois um SGBD possui vantagens intangíveis sobre os sistemas de arquivo tradicionais, como facilidade de uso, consolidação de informação de uma grande empresa, disponibilidade mais ampla de dados e acesso mais rápido à informação. O acesso com base em Web pode tornar parte dos dados acessíveis a todos os empregados ou mesmo a usuários externos. Benefícios mais tangíveis incluem o custo reduzido para o desenvolvimento de aplicações, redução da redundância de dados, melhor controle e segurança. Embora os bancos de dados estejam fortemente inseridos na maioria das organizações, a decisão de migrar de uma aplicação arquivo-baseada para uma abordagem centrada em banco de dados ainda surge com freqüência. Essa migração geralmente ocorre pelos seguintes fatores:

1. *Complexidade de dados*: Como os relacionamentos entre os dados são mais complexos, a necessidade de um SGBD torna-se mais efetiva.

2. *Compartilhamento entre aplicações*: O maior compartilhamento de dados entre aplicações e a maior redundância entre os arquivos aumentam, conseqüentemente, a necessidade de um SGBD.

3. *Evolução dinâmica ou crescimento dos dados*: Como os dados são alterados constantemente, é mais fácil atender a essas mudanças usando um SGBD do que um sistema de arquivos.

4- *Freqüência de consultas ad hoc aos dados*: Os sistemas de arquivo não facilitam a recuperação de dados ad hoc. 5. *Volume de dados e necessidade de controle*: O volume de dados e a necessidade de controlá-los exigem um SGBD.

É difícil desenvolver um conjunto genérico de diretrizes que possam adotar uma única abordagem para a administração dos dados de uma organização — se relacional, orientado a objeto, ou objeto-relacional. Se os dados a serem armazenados no banco de dados tiverem um alto nível de complexidade e envolverem vários tipos de dados, normalmente se consideram as abordagens orientadas a objeto ou objeto-relacionais para os SGBDs. Além disso, os benefícios da herança de classes e as vantagens da reutilização contam a favor dessas abordagens. Finalmente, vários fatores econômicos e organizacionais afetam a escolha de um SGBD:

1. *Adoção massiva de uma certa filosofia na organização*: Este é, freqüentemente, um fator dominante que afeta a aceitabilidade de um certo modelo de dados (por exemplo, relacional contra o orientado a objeto), de um certo vendedor ou de uma certa metodologia e ferramentas de desenvolvimento (por exemplo, o uso da análise objeto-orientada e de metodologias e ferramentas de projeto pode ser necessário para as novas aplicações).

5 Veja discussão relativa a esse assunto no Capítulo 22.

272 Capítulo 12 Metodologia para Projeto Prático de Banco de Dados e Uso de Diagramas UML

2. *Familiaridade do pessoal com o sistema*: Se os programadores da organização estiverem familiarizados com um SGBD em particular, ele pode ser favorecido de forma a reduzir treinamento e tempo de aprendizagem.

3. *Disponibilidade de serviço de venda*: É importante a disponibilidade da ajuda de vendedores que resolvam problemas com o sistema, uma vez que a migração de um ambiente sem SGBD para um com SGBD geralmente é um grande empreendimento e requer, inicialmente, muita assistência.

Outro fator a considerar é a portabilidade do SGBD entre os diferentes tipos de hardware. Muitos SGBDs comerciais têm versões que atualmente rodam em diversas configurações de hardware/software (ou **plataformas**). A necessidade de aplicações para backup, recuperação, desempenho, integridade e segurança também deve ser considerada. Muitos SGBDs são atualmente projetados para *soluções totais* no processamento e na administração dos recursos de informações dentro das organizações. A maioria dos vendedores de SGBD está combinando seus produtos com as seguintes opções ou características:

- Editores de texto e navegadores (*browsers*).
- Geradores de relatórios e utilidades para emissão.
- Software de comunicação (frequentemente chamados de monitores de teleprocessamento).
- Funcionalidades para entrada de dados e dispositivos de apresentação, como formulários, telas e menus, com características de edição automática.
- Ferramentas de busca e acesso que podem ser usadas na World Wide Web (facilidades para Web).
- Ferramentas gráficas para projeto de banco de dados.

Uma grande quantidade de softwares para 'terceiros' deve estar disponível para proporcionar funcionalidades ao SGBD em cada uma das áreas anteriores. Em raras situações pode ser preferível desenvolver software *in-house* (doméstico) em vez de usar um SGBD — por exemplo, se as aplicações são muito bem definidas e *totalmente* conhecidas. Em tais circunstâncias, um sistema projetado e customizado *in-house* (desenvolvido pela equipe da empresa) pode ser apropriado para implementar as aplicações conhecidas de modo mais eficiente. Na maioria dos casos, porém, novas aplicações, não previstas em tempo de projeto, aparecem *após* a implementação do sistema. É precisamente por isso que os SGBDs tornaram-se tão populares: eles facilitam a incorporação de novas aplicações apenas com modificações incrementais ao projeto existente do banco de dados. Tal evolução de projeto — ou evolução de esquema — é uma característica presente, em graus variados, nos SGBDs comerciais.

12.2.4 Fase 4: Mapeamento do Modelo de Dados (Projeto Lógico de Banco de Dados)

A próxima fase do projeto de um banco de dados é criar um esquema conceitual e esquemas externos do modelo de dados de SGBD selecionado, por meio do mapeamento dos esquemas produzidos na Fase 2a. O mapeamento pode ocorrer em dois estágios:

1. *Mapeamento de sistema-independente*: Nesta fase, o mapeamento não considera nenhuma característica específica ou casos especiais que se aplicam à implementação do modelo de dados do SGBD. Já discutimos o mapeamento SGBD-independente de um esquema ER para um esquema relacional na Seção 7.1, e esquemas EER para esquemas relacionais na Seção 7.2.

2. *Adaptando esquemas a um SGBD específico*: SGBDs diferentes implementam um modelo de dados usando características específicas de modelagem e restrições. Podemos ajustar os esquemas, obtidos no Passo 1, para conformarem-se às características de implementação específicas do modelo de dados usado pelo SGBD selecionado.

O resultado desta fase deveria ser declarações DDL na linguagem do SGBD escolhido, que especificam os esquemas de nível conceitual e externo do sistema de banco de dados. Entretanto, se as declarações DDL incluem alguns parâmetros do projeto físico, a especificação DDL completa terá de esperar até que a fase do projeto físico do banco de dados seja completada. Muitas ferramentas CASE de projeto (*computer-assisted software engineering* — engenharia de software auxiliada por computador — Seção 12.5) podem gerar DDLs para sistemas comerciais a partir do projeto conceitual de um esquema.

12.2.5 Fase 5: Projeto Físico do Banco de Dados

Projeto físico do banco de dados é o processo de escolha das estruturas de armazenamento específicas e dos caminhos de acesso para os arquivos de banco de dados, de modo a alcançar um bom desempenho nas várias aplicações de banco de dados. Cada SGBD oferece uma variedade de opções para a organização dos arquivos e caminhos de acesso. Essas opções normalmente incluem vários tipos de índices, agrupamento de registros relacionados em blocos de disco, ligações de registros via ponteiros e vários tipos de *hashing* (*separação em partes menores*). Uma vez escolhido um SGBD específico, o processo de projeto físico

12.2 O Processo de Projeto e Implementação de Banco de Dados

273

do banco de dados é restrito à escolha das estruturas mais apropriadas para os arquivos de banco de dados dentre as opções oferecidas por aquele SGBD. Nesta seção daremos diretrizes genéricas para essas decisões, adequadas a qualquer tipo de SGBD. Os seguintes critérios freqüentemente orientam a escolha de opções do projeto físico do banco de dados:

1. *Tempo de resposta*: É o tempo decorrente a partir da submissão de uma transação para o banco de dados, sua execução e resposta. A principal influência para o tempo de resposta, sob controle do SGBD, é o tempo de acesso do banco de dados para referenciar os itens de dados para a transação. O tempo de resposta também é influenciado por fatores fora do controle do SGBD, como a carga do sistema, a programação do sistema operacional ou os atrasos de comunicação.
2. *Utilização de espaço*: É o espaço total de armazenamento usado pelos arquivos do banco de dados e por suas estruturas de caminho de acesso em disco, incluindo índices e outros caminhos de acesso.
3. *Taxa de processamento (throughput) de transações*: É a média do número de transações que podem ser processadas por minuto; é um parâmetro crítico em sistemas de transações bancárias ou de reservas de linhas aéreas. A taxa de processamento de transações deve ser medida sob as condições de pico do sistema.

Normalmente, a média e o caso crítico são especificados, nos parâmetros precedentes, como parte dos requisitos de desempenho do sistema. São usadas técnicas analíticas ou experimentais, que podem incluir prototipação e simulação, para calcular os valores médios e críticos sob diferentes projetos físicos, e para determinar se eles satisfazem os requisitos de desempenho especificados. O desempenho depende do tamanho do registro e do número de registros no arquivo. Conseqüentemente, devemos estimar esses parâmetros para cada arquivo. Além disso, deveríamos calcular os padrões de crescimento das atualizações e das consultas, nos arquivos, para todas as transações. Atributos usados para selecionar registros deveriam ter caminhos de acesso primários e índices secundários construídos para eles. Estimativas de crescimento de arquivo ou do tamanho do registro, por causa de novos atributos ou do número de registros, também deveriam ser consideradas durante o projeto físico do banco de dados.

O resultado da fase de projeto físico do banco de dados é a determinação *inicial* das estruturas de armazenamento e dos caminhos de acesso para os arquivos do banco de dados. Quase sempre é necessário modificar o projeto, com base no desempenho observado, depois da implementação do sistema de banco de dados. Incluiremos essa atividade de **sintonização (tuning) do banco de dados** na próxima fase, e cobriremos esse tópico dentro do contexto de otimização de consultas no Capítulo 16.

12.2.6 Fase 6: Implementação e Sintonização do Sistema de Banco de Dados

Depois que os projetos lógicos e físicos são completados, podemos implementar o sistema de banco de dados. Essa responsabilidade é normalmente do DBA, e é levada a cabo com os projetistas. Declarações DDL (linguagem de definição de dados), inclusive SDL (linguagem de definição de armazenamento), do SGBD selecionado, são compiladas e usadas para criar o esquema do banco de dados (vazio) e os arquivos. O banco de dados pode ser, então, **carregado** com os dados. Se os dados forem convertidos de outro sistema, podem ser necessárias **rotinas de conversão** para reformatá-los antes de carregá-los no novo banco de dados.

As transações do banco de dados devem ser implementadas pelos programadores de aplicações, a partir das especificações conceituais das transações, escrevendo e testando o código dos programas com comandos DML embutidos. Uma vez prontas as transações, e os dados já carregados no banco de dados, terminam as fases de projeto e implementação e começa a fase operacional do sistema de banco de dados.

A maioria dos sistemas inclui monitoramento para estatísticas de desempenho, que são mantidas no catálogo do sistema ou no dicionário de dados para análise posterior. Essas estatísticas incluem no número de chamadas de transações para consultas predefinidas, atividade de entrada/saída em arquivos, contagem de páginas de arquivos ou registros de índice e freqüência de uso de índices. Como os requisitos do sistema de banco de dados mudam, freqüentemente é necessário adicionar ou remover tabelas e reorganizar alguns arquivos, mudando métodos de acesso primários, excluindo ou definindo novos índices.

Para melhorar o desempenho, podem ser reescritas algumas consultas ou transações. A sintonização do banco de dados continua ao longo de sua existência, considerando que possam ser descobertos problemas de desempenho e que os requisitos possam mudar.

12.3 Uso DE DIAGRAMAS UML COMO APOIO PARA A ESPECIFICAÇÃO DE PROJETO DE BANCO DE DADOS⁶

12.3.1 UML como um Padrão de Especificação de Projeto

Na primeira seção deste capítulo, discutimos em detalhes como as organizações trabalham com sistemas de informação e elaboram as várias atividades no ciclo de vida do sistema de informação. Os bancos de dados são parte integrante dos sistemas de informação na maioria das organizações. As fases de projeto de banco de dados se iniciam na análise de requisitos e seguem até a implementação do sistema; a sintonização foi introduzida no final da Seção 12.1 e discutida em detalhes na Seção 12.2. A indústria sempre tem necessidade de fazer algumas aproximações padrão para cobrir esse espectro inteiro de análise de requisitos, além da modelagem, do projeto, da implementação e da manutenção. A abordagem que está recebendo grande atenção e aceitabilidade, e que também é proposta como padrão pelo OMG (Grupo de Administração de Objeto), é a UML — **Unified Modeling Language** (Linguagem Unificada de Modelagem). Ela proporciona um mecanismo, na forma de notação diagramática, associado a uma sintaxe de linguagem para cobrir todo o ciclo de vida. Atualmente, a UML é usada pelos desenvolvedores de software, modeladores e projetistas de dados, arquitetos de banco de dados etc. para definir a especificação detalhada de uma aplicação. Eles também a usam para especificar o ambiente, que consiste em software, comunicação e hardware, e para implementar e manter a aplicação.

A UML combina conceitos aceitos de muitos métodos de *OO* e metodologias (notas bibliográficas para as metodologias que contribuíram para orientar a UML). Ela é aplicável a qualquer domínio e linguagem — é plataforma-independente; assim, os arquitetos de software podem modelar, em UML, qualquer tipo de aplicação, rodando em qualquer sistema operacional, linguagem de programação ou rede, tornando a abordagem amplamente aplicável. Ferramentas como a Rational Rose são comuns hoje para desenhar os diagramas da UML — elas permitem ao desenvolvedor de software criar modelos claros e inteligíveis para especificação, visualização, construção e documentação de componentes dos sistemas de software. Como a UML apoia extensamente o desenvolvimento de software e aplicações, não a cobriremos integralmente aqui. Nossa meta é mostrar algumas das notações pertinentes da UML, que são geralmente usadas na coleta e na análise de requisitos, também nas fases do projeto conceituai (fases 1 e 2 da Figura 12.1). A metodologia de desenvolvimento de aplicação detalhada usando a UML está fora do objetivo deste livro, mas pode ser encontrada em vários livros didáticos dedicados a projeto orientado a objeto, criação de software e UML (notas bibliográficas).

Os diagramas de classe, que são o resultado do projeto conceituai do banco de dados, já foram discutidos nas seções 3.8 e 4-6. Para chegar aos diagramas de classe, podem ser colhidas e especificadas as informações por meio dos diagramas de caso de uso, diagrama de sequência e diagramas de estados. Até o fim desta seção, introduziremos brevemente os diferentes tipos de diagramas UML para dar ao leitor uma idéia de sua extensão. Apresentaremos, em seguida, uma pequena aplicação-exemplo para ilustrar a utilização dos casos de uso, sequência e estados, e para mostrar como eles eventualmente orientam o diagrama de classe, bem como o projeto conceituai final. Os diagramas apresentados nesta seção pertencem à notação UML padrão e foram confeccionados pela ferramenta Rational Rose. A Seção 12.4 será dedicada a uma discussão geral do uso da Rational Rose em projetos de aplicação de banco de dados.

12.3.2 A UML para Projeto de Aplicação de Banco de Dados

A comunidade de banco de dados passou a adotar a UML, e atualmente muitos projetistas de banco de dados e desenvolvedores estão usando a UML para modelagem de dados, bem como nas fases subsequentes do projeto. A vantagem da UML é que, embora seus conceitos estejam baseados em técnicas orientadas a objeto, os modelos resultantes de estrutura e comportamento podem ser usados para projetar tanto bancos de dados relacionais quanto orientados a objeto e objeto-relacionais (capítulos 20 a 22 — definição de banco de dados orientado a objeto e objeto-relacional). Já dissemos que os **Diagramas de Classe** da UML são semelhantes aos de ER e EER das seções 3.8 e 4-6, respectivamente. Eles fornecem uma especificação estrutural do esquema de banco de dados em um senso orientado a objeto, mostrando o nome, os atributos e as operações de cada classe. Sua função geral é descrever a coleção de objetos de dados e seus relacionamentos, o que é compatível com a meta do projeto conceituai de banco de dados.

6 A contribuição de Abrar Ul-Haque para as seções de UML e Rational Rose foi muito importante.

12.3 Uso de Diagramas UML como Apoio para a Especificação de Projeto de Banco de Dados 275

Uma das principais contribuições da abordagem UML foi juntar os profissionais tradicionais de modelagem do banco de dados, os analistas e os projetistas aos desenvolvedores de aplicações de software. Na Figura 12.1 mostramos as fases de projeto e implementação de banco de dados e como elas se aplicam a esses dois grupos. A UML pôde propor uma notação comum, ou um metamodelo, que pode ser adotada por ambas as comunidades e adaptada às suas necessidades. Embora tenhamos nos dedicado somente ao aspecto estrutural da modelagem nos capítulos 3 e 4, a UML também nos permite fazer a modelagem estrutural e/ou comportamental, introduzindo vários tipos de diagramas. Como resultado, tem-se uma especificação/descrição mais completa da aplicação global de banco de dados. Nas próximas seções, resumiremos primeiro os diferentes diagramas da UML e, então, daremos um exemplo de caso de uso, seqüência e estado em uma aplicação. Um estudo de caso completo do desenvolvimento de aplicação de banco de dados é apresentado no Apêndice B, no site deste livro.

12.3.3 Os Diferentes Diagramas da UML

A UML define nove tipos de diagramas, divididos em duas categorias.

Diagramas estruturais. Descrevem as relações estruturais ou estáticas entre componentes. Compreendem o Diagrama de Classe, o Diagrama de Objeto, o Diagrama de Componentes e o Diagrama de Desenvolvimento.

Diagramas de comportamento. Seu propósito é descrever as relações de comportamento ou a dinâmica entre componentes.

Compreendem o Diagrama de Caso de Uso, o Diagrama de Seqüência, o Diagrama de Colaboração, o Diagrama de Estado e o Diagrama de Atividade.

Faremos, abaixo, uma breve apresentação dos nove tipos. Os diagramas estruturais compreendem:

A. Diagramas de Classe

Os diagramas de classe capturam a estrutura estática do sistema e servem de base para outros modelos. Eles mostram classes, interfaces, colaborações, dependências, generalizações, associação e outros relacionamentos. Os diagramas de classe são muito úteis para modelagem do esquema conceitual de banco de dados. Mostramos exemplos de diagramas de classe para o esquema de banco de dados de uma empresa na Figura 3.16 e para uma hierarquia de generalização na Figura 4-10.

Diagramas de pacote. Diagramas de pacote são subconjuntos dos diagramas de classe. Servem para organizar os elementos do sistema em grupos correlacionados chamados pacotes. Um pacote pode ser uma coleção de classes relacionadas e seus relacionamentos. Os diagramas de pacote ajudam a minimizar as dependências em um sistema.

B. Diagramas de Objeto

Os diagramas de objeto mostram um conjunto de objetos e seus relacionamentos. Eles correspondem ao que chamamos de diagramas de instâncias nos capítulos 3 e 4. Eles dão uma visão estática do sistema em um momento particular e normalmente são usados para testar a precisão dos diagramas de classe.

C. Diagramas de Componentes

Os diagramas de componentes ilustram a organização e as dependências entre os componentes de software. Um diagrama de componente consiste em componentes, interfaces e relacionamentos de dependência. Um componente pode estar em código-fonte, *run-time* (*tempo de execução*) ou executável. É um bloco físico no sistema e é representado por um retângulo com dois retângulos menores ou abas em seu lado esquerdo. Uma interface é um grupo de operações usado ou criado por um componente e normalmente é representado por um círculo pequeno. O relacionamento de dependência é usado para modelar o relacionamento entre dois componentes, e é representado por uma seta pontilhada que aponta o componente do qual ele depende. Para bancos de dados, o diagrama de componentes representa dados armazenados, tais como tabelas ou partições. Interfaces referem-se às aplicações que usam os dados armazenados.

D. Diagramas de Desenvolvimento

Os diagramas de desenvolvimento representam a distribuição de componentes (executáveis, bibliotecas, tabelas, arquivos) pela topologia de hardware. Eles descrevem os recursos físicos de um sistema, inclusive nodos, componentes e conexões, e são usados para mostrar a configuração dos elementos em tempo de execução (os nós) e os processos de software que neles residem (*threads*). Descreveremos, agora, os diagramas de comportamento, expandindo aqueles que forem de interesse particular.

E. Diagramas de Caso de Uso

Os diagramas de caso de uso são usados para modelar as interações funcionais entre os usuários e o sistema. Um cenário é uma sucessão de passos que descrevem uma interação entre um usuário e o sistema. Um caso de uso é um conjunto de cenários que

Capítulo 12 Metodologia para Projeto Prático de Banco de Dados e Uso de Diagramas UML

têm uma meta comum. O diagrama de caso de uso foi introduzido por Jacobson para visualização dos casos de uso. O diagrama de caso de uso mostra os atores interagindo e pode ser facilmente entendido, mesmo sem conhecimento de nenhuma linguagem. Um caso de uso em particular é representado por uma elipse e representa uma tarefa específica executada pelo sistema. Um ator, simbolizado por uma pessoa com uma seta, representa um usuário externo que pode ser uma pessoa, um grupo representativo de usuários, um papel representado por uma pessoa na organização, ou qualquer coisa externa ao sistema. O diagrama de caso de uso mostra as interações possíveis do sistema (em nosso caso, um sistema de banco de dados) e descreve como os casos de uso executam tarefas específicas do sistema. Como eles não especificam nenhum detalhe de implementação e muito fáceis de entender, são um bom veículo de comunicação entre os usuários finais e os desenvolvedores, e agilizam a validação do usuário. Planos de teste podem ser gerados facilmente usando os diagramas de caso de uso. A Figura 12.7 mostra a notação do diagrama de caso de uso. O relacionamento *include* é usado para fatorar um comportamento comum entre dois ou mais casos de uso originais — é uma forma de reuso. Por exemplo, no ambiente universitário mostrado na Figura 12.1, caso de uso 'matrícula em curso' e 'informa grade', nos quais aluno e professor são os atores envolvidos, incluem um caso de uso comum chamado 'valida usuário'. Se um caso de uso incorpora dois ou mais cenários significativamente diferentes, baseados em circunstâncias ou condições variadas, o relacionamento *extend* é usado para mostrar os subcasos vinculados ao caso básico (Figura 12.7).

Diagramas de interação. Os diagramas de interação são usados para modelar os aspectos dinâmicos de um sistema. Eles consistem basicamente em um conjunto de troca de mensagens entre um conjunto de objetos. Há dois tipos de diagramas de interação:

Sequência e Colaboração.

Caso de Uso Básico _1

«include»

Caso de Uso Incluído

«include»

Ator _1

Caso de Uso Básico _2

Caso de Uso

Ator _2

Ator 4

«extend» -

Caso de Uso Estendido

Caso de Uso Básico _3

Ator _3 **FIGURA 12.7** A notação do diagrama de caso de uso.

7 Veja Jacobson et al. (1992).

12.3 Uso de Diagramas UML como Apoio para a Especificação de Projeto de Banco de Dados 277

«include»

Matrícula em Curso

" ~ ^ .._____

L_J

, "" Valida Usuário

«include»

Professor

Solicita Auxílio

Diretoria de Auxílio Financeiro

FIGURA 12.8 Um exemplo de diagrama de caso de uso para um banco de dados universitário.**F. Diagramas de Seqüência**

Os diagramas de seqüência descrevem as interações entre os vários objetos ao longo do tempo. Eles fornecem uma visão dinâmica do sistema mostrando basicamente o fluxo de mensagens entre os objetos. Em um diagrama de seqüência, um objeto ou um ator é mostrado como uma caixa no topo de uma linha vertical, chamada **linha da vida do objeto**. Para um banco de dados, esse objeto é normalmente alguma coisa física (como um livro em uma livraria), que poderia estar contida no banco de dados: um documento externo ou formulário, como um impresso para pedidos, ou uma tela para apresentação, que pode ser parte de uma interface com o usuário. A linha da vida representa a existência do objeto ao longo do tempo. A **ativação** indica quando um objeto está executando uma ação e é representada por uma caixa retangular na linha da vida. Cada mensagem é representada por uma seta entre as linhas da vida de dois objetos. Uma mensagem ostenta um nome e pode ter argumentos e informações de controle para explicar a natureza da interação. A ordem de leitura das mensagens é de cima para baixo. Um diagrama de seqüência também dá opção para autochamada, que é basicamente uma mensagem de um objeto para si próprio. Também podem ser mostrados no diagrama de seqüência **condições** e **marcadores** de repetição, a fim de especificar quando uma mensagem deve ser enviada e a condição de envio com múltiplos marcadores. Uma linha de retorno mostra uma mensagem de retorno e é opcional, a menos que tenha um significado especial. A eliminação de um objeto é mostrada com um X. A Figura 12.9 explica a notação do diagrama de seqüência.

G. Diagramas de Colaboração

Os diagramas de colaboração representam interações entre objetos como séries de mensagens seqüenciadas. Nos diagramas de colaboração, a ênfase está na organização estrutural dos objetos que enviam e recebem mensagens, enquanto nos diagramas de seqüência a ênfase está na ordenação temporal das mensagens. Os diagramas de colaboração mostram os objetos como ícones e numeram as mensagens; a numeração das mensagens representa sua ordenação. O plano espacial do diagrama de colaboração permite acoplamentos entre objetos que mostram seus relacionamentos estruturais. Usar o diagrama de colaboração ou o de seqüência para representar interações é uma questão de escolha; usaremos apenas os diagramas de seqüência a partir de agora.

H. Diagramas de Estado

Os diagramas de estado descrevem as mudanças de estado de um objeto em relação a eventos externos.

Para descrever o comportamento de um objeto é comum, na maioria das técnicas de orientação a objeto, criar um diagrama de estado para mostrar todos os possíveis estados que um objeto pode assumir durante sua existência. O diagrama de estado da UML é baseado no diagrama de estado de David Harel. Ele mostra uma máquina de estados constituída basicamente de estados, transições, eventos e ações, e é muito útil no projeto conceitual de aplicações que trabalham com objetos armazenados no banco de dados.

8 Veja Harel (1987).

278

Capítulo 12 Metodologia para Projeto Prático de Banco de Dados e Uso de Diagramas UML

Objeto:Classe ou AtorObjeto:Classe ou Ator**D****D****Tempo de Vida** i**Mensagem****Foco de Controle/Ativação****Lf****:m**

Automensagem

5**X**Objeto:Classe ou Ator**D**

i

D

i

Objeto Destruição/Término

FIGURA 12.9 A notação do diagrama de sequência.

Os elementos importantes de um diagrama de estado, mostrados na Figura 12.10, são os seguintes:

- Estados: mostrados como caixas com cantos arredondados, representam situações na vida de um objeto.
- Transições: mostradas como setas sólidas entre os estados, representam os caminhos entre os estados diferentes de um objeto. São rotuladas pelo nome do evento [guarda]/ação; o evento ativa a transição e a ação é o seu resultado. *Guarda* uma condição adicional e opcional que especifica uma condição sob a qual a mudança de estado pode não acontecer
- Start/Estado Inicial: mostrado por um círculo sólido, com uma seta de partida para um estado.
- Stop/Estado Final: mostrado por um círculo cheio com linha dupla, e com uma seta vinda de um estado, apontando para ele.

Os diagramas de estado são úteis para especificar como a reação de um objeto a determinada mensagem depende de seu estado. Um evento é algo feito a um objeto, como o envio de uma mensagem; uma ação é algo que um objeto faz, como enviar uma mensagem.

I. Diagramas de Atividade

Os diagramas de atividade apresentam uma visão dinâmica do sistema, modelando o fluxo de controle de uma atividade para outra. Podem ser considerados fluxogramas com estados. Uma atividade é um estado de 'fazer alguma coisa', que poderia ser um processo do mundo real ou uma operação em alguma classe do banco de dados. Normalmente, os diagramas de atividade são usados para modelar o fluxo de trabalho e as operações empresariais internas para uma aplicação.

12.3.4 Um Exemplo de Modelagem e Projeto: Banco de Dados Universitário

Nesta seção ilustraremos brevemente o uso dos diagramas da UML, apresentando o projeto de um banco de dados relacionado para controle universitário. Será omitido um grande número de detalhes para economia de espaço; só o uso genérico desses diagramas, que conduzirão ao projeto conceitual e ao projeto dos componentes do programa, será ilustrado. Como afirmamos antes, o SGBD final, no qual esse banco de dados será implementado, pode ser relacional, orientado a objeto ou objeto-relacional. Isso não mudará a análise e a modelagem da aplicação que usa os diagramas UML.

Imagine um cenário em que alunos se matriculem em cursos que são oferecidos por professores. A secretaria é encarregada da programação dos cursos e da manutenção de um catálogo sobre eles. Ela tem autoridade para adicionar e cancelar cursos, além de promover alterações. Também são fixados limites de matrículas para os cursos. A diretoria de ajuda financeira processa as aplicações de apoio aos alunos, à qual eles devem solicitar auxílio. Suponha que temos de projetar um banco de dados.

12.3 Uso de Diagramas UML como Apoio para a Especificação de Projeto de Banco de Dados

279

dados que mantêm os dados sobre alunos, professores, cursos, ajuda etc. Também queremos projetar uma aplicação que nos permita fazer a inscrição no curso, o processamento do formulário de ajuda financeira e a manutenção de um grande catálogo de cursos da universidade na secretaria de registro. Os requisitos anteriores podem ser descritos por uma série de diagramas UML, como mostraremos adiante.

-----1\

Start/Estado Inicial

Transição

Estado

Estado 2

Estado 3

1

...

Estado consiste em três partes
 Nome Atividades Máquina
 Embutida
 Atividades e Máquina
 Embutida são opcionais

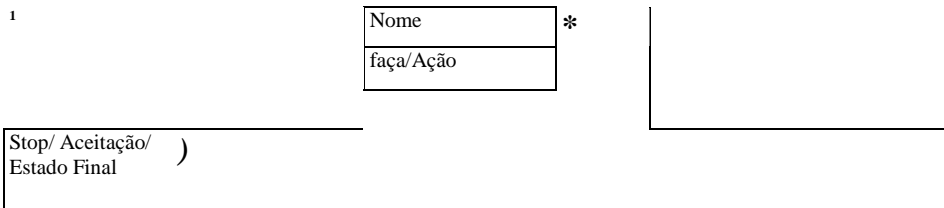


FIGURA 12.10 A notação do diagrama de estado.

Como previamente mencionado, um dos primeiros passos envolvidos no projeto de um banco de dados é a coleta dos requisitos do cliente, e o melhor modo para fazer isso é usar o diagrama de caso de uso. Suponha que um dos requisitos do banco de dados universitário seja permitir que os professores entrem com a grade dos cursos que irão ministrar, que os alunos possam se matricular nos cursos e solicitar ajuda financeira. O diagrama de caso de uso, que corresponde a esses casos, pode ser desenhado como mostra a Figura 12.8.

É útil também, quando se projeta um sistema, representar graficamente alguns dos estados que o sistema pode assumir. Isso ajuda a visualização dos vários estados do sistema durante o curso da aplicação. Por exemplo, em nosso banco de dados universitário, os vários estados pelos quais o sistema passa quando é aberta a inscrição para um curso com 50 vagas podem ser representados pelo diagrama de estado da Figura 12.11. Note que são mostrados os estados de um curso enquanto as matrículas estão abertas. Durante o estado de matrícula, a transição 'Matricula Aluno' continua enquanto o número de alunos matriculados for menor que 50.

Prontos os diagramas de caso de uso e de estado, podemos fazer o diagrama de sequência para visualizar a execução dos casos de uso. Para o banco de dados universitário, o diagrama de sequência corresponde ao caso de uso: o aluno faz inscrição e seleciona um curso em particular para se matricular — isso é mostrado na Figura 12.12. São conferidos os pré-requisitos e a capacidade do curso;

então o curso é acrescentado ao horário do aluno se os pré-requisitos forem atendidos e se houver vaga. 280

Capítulo 12 Metodologia para Projeto Prático de Banco de Dados e Uso de Diagramas UML

Matricule Aluno [contador < 50]

Matricula em Curso faz/matricula alunos

Matricula Aluno/marque contador = 0

Matriculando entra/registra aluno

Cancelamento

contador = 50

Cancelamento

Cancelado

Cancelamento

Encerramento de Matrícula término/encerramento matrícula

FIGURA 12.11 Um exemplo de diagrama de estado para o banco de dados universitário.

:Aluno

solicitaInscricao

Inscrição :Catalogo

:Curso

obtemListaCursos

selecionaCurso

adicionaCurso

:Horario

obtemPreReq

obtemVagas I

obtemPreq = verdadeiro && [obtemVagas = Verdadeiro/atualizaHorario

!D

FIGURA 12.12 Diagrama de sequência para o banco de dados universitário.

Os diagramas UML anteriores *não* são a especificação completa do banco de dados universitário. Haverá outros c de uso Com o registrador como ator, ou o aluno comparecendo a um exame e colando grau no curso etc. Uma metodoL completa de como chegar aos diagramas de classe, a partir dos diversos diagramas ilustrados acima, está fora de nossa pre são aqui. Isso será explicado mais adiante, no estudo de caso (Apêndice B, no site deste livro). Metodologias de projeto ai são sujeitas a julgamentos, preferências pessoais etc, porém, podemos ter certeza de que o diagrama de classe responderá todas as especificações que foram determinadas pelos casos de uso, diagramas de estado e sequência. O diagrama de classe

281

Figura 12.13, mostra as classes com seus relacionamentos estruturais e as operações dessas classes, que foram derivadas desses diagramas. Essas classes precisarão ser implementadas para desenvolver o banco de dados da universidade e, com as operações, implementarão a aplicação completa da classe horário/matricula/auxílio. Para melhor compreensão, somente os atributos mais importantes são mostrados nas classes, com certos métodos originados pelos diagramas mostrados. É concebível que esses diagramas de classe sejam constantemente atualizados, e mais detalhes sejam especificados e mais funções envolvidas na aplicação universitária.

EMPREGADO

D Fnome:CHAR(15)

□ InicialM:CHAR(1)

D Lnome :CHAR(15)

D Sexo:CHAR(1)

D Salário : INTEGER

D Endereço : CHAR(20)

B Ssn : INTEGER

D DataNasc : DATE

H Numero : INTERGER

H Numero_Projeto : INTEGER

@ Nome:CHAR(15)

@ EMPREGADO_Ssn : INTEGER

0 «PK» PK_T_00() [3 «FK» EMPREGAD02() [3 «FK» EMPREGADO60 g «FK» EMPREGADO10()

T

«Non-Identifying»

WORKS_FOR 1..* 1

0..1*

1 «Non-Identifying»

*MANAGES***0..1***

DEPARTAMENTO

0 Numero : INTEGER

D Nome:CHAR(15)

□ Localização : CHAR(15)

D Nro.,de_employado : INTEGER

D GerSsn : INTEGER

D GRDataInicio : DATE

@ Ssn : INTEGER

«PK» PK_DEPARTAMENTO01() «FK» FKJ3EPARTAMENTO07() «Unique» TC_DEPARTAMENTO024()

I¹

«Identifying»

TEM^DEPENDENTES

0..*

«NON-Identifying» *SUPERVISIONA*****«Identifying» *TRABALHA_PARA*«NON-Identifying» *CONTROLA*

0..*

DEPENDENTE

0 Nome:CHAR(15)

D SEXO:CHAR(1)

D DataNasc: DATE

D Parentesco : CHAR(15)

M Ssn : INETGER

g «PK» PK_DEPENDENTE3() g «FK» FK_DEPENDENTE1()

PROJETO

0 Numero : INTEGER

0 Nome:CHAR(15)

D Localização : CHAR(15)

0 Numero_DEPARTAMENTO : INTEGER

D Horas :TIME(2)

«PK» PK_PROJETO02() «FK» FK_PROJETO03()

FIGURA 12.13 Um diagrama gráfico de modelo de dados em Rational Rose.

12.4 RATIONAL ROSE, UMA FERRAMENTA DE PROJETO BASEADA EM UML

12.4.1 Rational Rose para Projeto de Banco de Dados

A Rational Rose é uma das ferramentas de modelagem mais importantes usadas na indústria para o desenvolvimento de sistemas de informação. Como mostramos nas primeiras duas seções deste capítulo, o banco de dados é um componente central da maioria dos sistemas de informação e, conseqüentemente, a Rational Rose possibilita a especificação inicial em UML que, no final, conduz ao desenvolvimento do banco de dados. Foram feitas muitas extensões para modelagem de dados nas mais recentes versões da Rose, e atualmente ela dá suporte para projeto e modelagem conceituai, lógica e física do banco de dados.

12.4.2 Rational Rose Data Modeler

Rational Rose Data Modeler é uma ferramenta de modelagem visual para bancos de dados. Uma das razões para sua popularidade é que, diferentemente das demais ferramentas de modelagem de dados, ela é baseada em UML. Seu uso proporciona uma ferramenta comum e uma linguagem para cobrir falhas de comunicação entre projetistas de banco de dados e desenvolvedores de aplicação; possibilita também o trabalho conjunto de projetistas de banco de dados, desenvolvedores e analistas, na captura e no compartilhamento dos requisitos empresariais, acompanhando suas alterações ao longo do processo. Também permite que os projetistas modelem e projetem todas as especificações na mesma plataforma, que usa a mesma notação — isso melhora o processo de projeto e reduz o risco de erros.

Outra importante vantagem da Rose é sua capacidade de processo de modelagem que permite modelar o comportamento do banco de dados, como vimos no pequeno exemplo acima, na forma de diagramas de caso de uso, sequência e estado. Há ainda os diagramas de colaboração, para mostrar interações entre objetos, e o diagrama de atividade para modelar o fluxo de controle, que não elaboramos. A meta é, tanto quanto possível, gerar a especificação do banco de dados e o código da aplicação. Com a Rational Rose Data Modeler podemos definir gatilhos, procedimentos armazenados etc. explicitamente no diagrama (o Capítulo 24 mostra os bancos de dados ativos que contêm essas características) em vez de representá-los como valores escondidos atrás dos cenários. O Data Modeler também proporciona capacidade de engenharia avante no banco de dados, em termos de possibilitar atualizações constantes dos requisitos, e de engenharia reversa para o projeto conceituai de um banco de dados já implementado.

12.4.3 Modelagem de Dados Usando o Modelador de Dados da Rational Rose

Há muitas ferramentas e opções disponíveis para modelagem de dados no Rational Rose Modeler. O modelador de dados da Rational Rose permite criar um modelo de dados com base na estrutura do banco de dados, ou criar um banco de dados baseado no modelo de dados.

Engenharia Reversa. A engenharia reversa de um banco de dados permite que o usuário crie um modelo de dados baseado na estrutura de banco de dados. Se tivermos um banco de dados criado em um SGBD, ou nos arquivos DDL, podemos usar a ferramenta (*wizard*) de engenharia reversa do modelador de dados da Rational Rose para gerar um modelo de dados conceituai. O *wizard* de engenharia reversa lera o esquema básico do banco de dados, ou os arquivos DDL, e os transformará em um modelo de dados. Enquanto faz isso, incluirá também todos os nomes identificadores das entidades.

Engenharia Avante e Gerador de DDL. Poderemos criar também, a partir do zero, um modelo de dados na Rational Rose. Criado o modelo de dados, podemos usá-lo para gerar DDLs em um SGBD específico. Há um *wizard* de engenharia avante no modelador que lê o esquema no modelo de dados ou lê ambos, o esquema do modelo de dados e as tabelas no modelo de armazenamento de dados, e cria o código DDL apropriado, gerando um arquivo DDL. O *wizard* também proporciona a opção de gerar um banco de dados executando o arquivo DDL gerado.

Projeto Conceituai em Notação UML. Como mencionado anteriormente, uma das principais vantagens da Rose é viabilizar a modelagem do banco de dados usando notação UML. O diagrama ER, freqüentemente usado no projeto conceituai de bancos de dados, pode ser facilmente construído usando a notação UML, por meio dos diagramas de classe da Rational Rose. Por exemplo, o esquema ER do nosso exemplo de empresa do Capítulo 3 pode ser refeito na Rational Rose usando a notação UML como segue. Ele pode ser convertido em forma gráfica usando a opção de diagrama dos dados modelo em Rose.

Os diagramas correspondem, em parte, a um esquema relacional (lógico), apesar de estarem em nível conceituai. Eles mostram os relacionamentos entre tabelas via chave primária (PK) — chave estrangeira (FK). A identificação de relacionamentos especifica que uma tabela filha não pode existir sem a tabela pai (tabelas dependentes), embora os relacionamentos não-identificados especifiquem uma associação regular entre duas tabelas independentes. Para um melhor e mais claro entendimento, as chaves estrangeiras aparecem automaticamente com um dos atributos das entidades filhas. É possível atualizar um esquema diretamente em seu texto ou forma gráfica. Por exemplo, o relacionamento entre EMPREGADO e PROJETO, chamado TRABALHA_EM, pode ser deletado e, automaticamente, a Rose cuidará de todas as chaves estrangeiras etc. na tabela.

9 O termo modelo de dados, usado pelo Rational Rose Modeler, corresponde à nossa noção de aplicação modelo.

sar tanto o modelo de objetos quanto o de dados e ver como eles estão relacionados, e assim fazer escolhas melhores e mais embasadas de como construir os métodos de acesso aos dados. Há também a opção de usar o *Rational Rose Web Publisher* para permitir que os modelos, e os metadados sob esses modelos, estejam disponíveis a todos os grupos.

Pessoa

nome Ssn

L3 visaoHorario ()

L3()

AuxilioFinanceiro

Tipoauxilio Somaauxilio

Aluno

[3 atribuiAuxilio () L3 cancelaAuxilio ()

L3 solicitaInscricao () [3 aplicaAuxilio ()

L30

Professor

[3 entreGrades () [3 ofereceCurso ()

C3()

Inscrição

[3 encontraCursoAdicionar ()

[3 cancelaCurso ()

[3 adicionaCurso ()

[3 consultaHorario ()

0()

Horário

[3 atualizaHorario () [3 mostraHorario () **L3()**

Catalogo
B--
[3 obtemPreReq () [3 obtemVagas () [3 obtemListaCursos () L3()

”

Curso
IU duração IH sala B vagas m....
L3 cancelaCurso () L3 adicionaCurso () L3()

FIGURA 12.15 O projeto do banco de dados universitário como diagrama de classe.

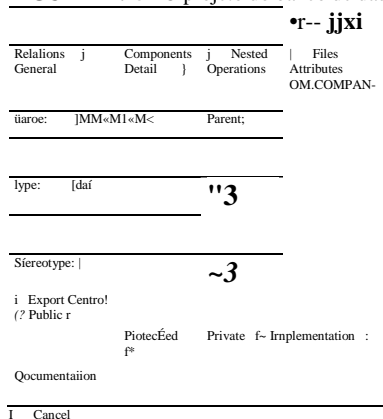


FIGURA 12.16 A classe OM_EMPLOYEE que corresponde à tabela empregado na Figura 12.14.

O que descrevemos até aqui corresponde apenas a uma visão parcial da capacidade da ferramenta, relacionada às fases de projeto conceituai e lógico da Figura 12.1. A gama inteira dos diagramas UML, que descrevemos na Seção 12.3, pode ser desenvolvida e mantida em Rose. Para detalhes adicionais, o leitor deve recorrer à literatura do produto. O Apêndice B, no site deste livro, desenvolve um estudo de caso completo com a ajuda dos diagramas UML e mostra o desenvolvimento do projeto em suas diferentes fases. A Figura 12.17 dá uma versão do diagrama de classe da Figura 3.16 usando a Rational Rose.

TRABALHA_PARA

EMPREGADO

m Pnome M Minic IS Unome M Ssn [§3 Datanasc l§3 Sexo l§1 Endereço H3 Salário

H idade ()

H alterajepartamento ()

H altera_projeto ()

Nome dependente -----G-----

n +supervi nona

0..1

+supervisor

ADMINISTRA

Datalnicio

DEPARTAMENTO

K>] Nome Numero

DEPENDENTE

Sexo

DataNasc

Relacionamento

TRABALHAR EM

Horas

Q adiciona_empregado () H nro_de_empregado () Q altera__chefia ()

00..n

1..n

LOCALIZAÇÃO

Nome

PROJETO

Nome Numero

[3 adicione_empregado () H adicione_projeto () [3 altera_gerente ()

0..n

FIGURA 12.17 Diagrama de classe para o banco de dados empresa (Figura 3.16) gerado pela Rational Rose.

12.5 FERRAMENTAS AUTOMATIZADAS PARA PROJETO DE BANCO DE DADOS

A atividade de projeto de banco de dados desenvolve, predominantemente, a Fase 2 (projeto conceituai), a Fase 4 (mapeamento do modelo de dados ou projeto lógico) e a Fase 5 (projeto físico do banco de dados) no processo de projeto que discutimos na Seção 12.2. A discussão da Fase 5 será adiada para o Capítulo 16, no contexto de otimização de consulta. Discutimos em detalhes as fases 2 e 4, com o uso da notação da UML na Seção 12.3, e pontuamos as características da ferramenta Rational Rose, que apoia essas fases. Como comentamos anteriormente, a Rational Rose é mais que uma ferramenta de projeto de banco de dados. Ela é também uma ferramenta de desenvolvimento de software e faz a modelagem do banco de dados e o esquema do projeto na forma de diagramas de classe, parte da metodologia global de desenvolvimento de uma aplicação orientada a objeto. Nesta seção resumiremos as características e as lacunas do conjunto de ferramentas comerciais focadas na automatização do processo de projeto conceituai, lógico e físico de bancos de dados.

Quando foi introduzida a tecnologia de banco de dados, a maioria dos projetos foi realizada manualmente por projetistas especializados, que usaram a sua experiência e o seu conhecimento no processo. Porém, pelo menos dois fatores indicavam que, se possível, alguma forma de automatização deveria ser utilizada:

1. Como as aplicações envolvem uma complexidade cada vez maior de dados, em termos de relacionamentos e restrições, o número de diferentes opções ou projetos para modelar a mesma informação aumenta rapidamente. Fica difícil de lidar manualmente com toda essa complexidade e com as correspondentes alternativas de projeto.
2. O tamanho total de alguns bancos de dados pode chegar a centenas de tipos entidade e relacionamento, fazendo com que a tarefa de administrar manualmente esses projetos seja quase impossível. As metas relacionadas ao processo de projeto, que descrevemos na Seção 12.2, rendem outro banco de dados a ser criado, mantido e consultado, como um banco de dados propriamente dito.

Os fatores anteriores deram origem a muitas ferramentas de projeto de banco de dados no mercado, que são colocadas na categoria geral de CASE (*Computer-Aided Software Engineering* — Engenharia de Software Apoiada por Computador). A Rational Rose é um bom exemplo de uma ferramenta moderna CASE. Normalmente essas ferramentas consistem em uma combinação das seguintes instalações:

1. *Diagramação*: Permite que o projetista crie um diagrama do esquema conceitual em alguma notação específica da ferramenta. A maioria das notações compreende tipos entidade, tipos relacionamento, que são mostrados como re-tângulos separados ou simplesmente como linhas ou setas; restrições de cardinalidade são mostradas ao lado das linhas ou em diferentes tipos de flechas ou como restrições min/max, atributos, chaves e assim por diante. Algumas ferramentas exibem hierarquias de herança e uso de notação adicional para mostrar a natureza das generalizações: parciais ou totais, desjuntas ou sobrepostas. Os diagramas são armazenados internamente, como projetos conceituais, e estão disponíveis para alterações ou para geração de relatórios, listagem das referências cruzadas e outros.
2. *Modelo de mapeamento*: Implementa o algoritmo de mapeamento semelhante ao que apresentamos nas seções 9.1 e 9.2. O mapeamento é mais específico do sistema — a maioria das ferramentas gera esquemas em DDL SQL para Oracle, DB2, Informix, Sybase e outros SGBDRs. Essa parte da ferramenta é mais tranquila para automatização. Se necessário, o projetista poderá editar os arquivos DDL produzidos.
3. *Normalização do projeto*: Utiliza um conjunto de dependências funcionais, que são fornecidas ao projeto conceitual ou depois que forem produzidos os esquemas relacionais durante o projeto lógico. Os algoritmos de decomposição de projeto do Capítulo 15 são aplicados para decompor as relações existentes em relações de formas normais mais altas. Em geral, as ferramentas falham ao não fornecer alternativas para a geração de projetos na 3FN ou na BCNF, de modo a permitir que o projetista possa fazer a seleção entre eles com base em algum critério como o número mínimo de relações ou a menor quantidade de armazenamento. A maioria das ferramentas incorpora alguma forma de projeto físico, inclusive a escolha de índices. Existe toda uma gama de ferramentas separadas para monitoramento e avaliação de desempenho. O problema para sintonização de um projeto, ou para a implementação de banco de dados, ainda é essencialmente uma atividade controlada manualmente. Além das fases de projeto descritas neste capítulo, uma área em que quase não há apoio de ferramentas comerciais é a integração de visão (Seção 12.2.2). Não analisaremos ferramentas de projeto de banco de dados aqui, apenas mencionaremos algumas características que uma boa ferramenta de projeto deve possuir:

1. *Uma interface fácil de usar*: Este fator é crítico, pois permite que os projetistas focalizem a tarefa, não o entendimento da ferramenta. Interfaces gráficas e ponto e clique são normalmente usadas. Poucas ferramentas, como a ferramenta de SECSI, usada na França, utilizam linguagem natural como entrada. Diferentes interfaces podem ser adaptadas a projetistas novatos ou especializados.
2. *Componentes analíticos*: As ferramentas deveriam fornecer componentes analíticos para tarefas que são difíceis de executar manualmente, como avaliar alternativas de projetos físicos ou detectar restrições contraditórias entre as visões. Esta área é fraca na maioria das ferramentas atuais.
3. *Componentes heurísticos*: Aspectos de projeto, que não podem ser quantificados precisamente, podem ser automatizados usando regras heurísticas, na ferramenta de projeto, para avaliar alternativas de projeto.
4. *Análise de intercâmbio*: Uma ferramenta deveria apresentar ao projetista uma análise comparativa adequada sempre que houvesse diversas alternativas de escolha. De forma ideal, as ferramentas deveriam incorporar uma análise dos efeitos das alterações de projeto, desde o projeto conceitual até o projeto físico. Por causa das muitas alternativas possíveis para o projeto físico em um determinado sistema, tal análise de intercâmbio é difícil de realizar, e a maioria das ferramentas atuais a evita.
5. *Exibição dos resultados de projeto*: Frequentemente são exibidos, em forma diagramática, resultados de projeto, como os esquemas. Diagramas bem dispostos e que agradem esteticamente não são fáceis de ser gerados automaticamente. Projetos com diversos planos, fáceis de ler, são outro desafio. Outros tipos de resultados de projeto, como tabelas, listas ou relatórios, que são de fácil interpretação, também podem ser apresentados.
6. *Verificação de projeto*: Esta é uma característica altamente desejável. Seu propósito é verificar se o projeto resultante satisfaz os requisitos iniciais. A menos que os requisitos sejam capturados e representados internamente em alguma forma analisável, a verificação não pode ser empreendida.

10 Mostramos as notações dos diagramas de classe ER, EER e UML nos capítulos 3 e 4- Veja o Apêndice A para ter uma idéia dos diferentes tipos de notações diagramáticas usados.

12.6 Resumo

287

Atualmente, há crescente consciência do valor das ferramentas de projeto, e elas estão se tornando um imperativo para lidar com os problemas de projeto de grandes bancos de dados. Também há uma conscientização cada vez maior de que o esquema do projeto e o projeto das aplicações deveriam seguir conjuntamente, e a tendência atual, entre as ferramentas CASE, é atender às duas áreas. A popularidade da Rational Rose deve-se ao fato de que ela se estende por dois ramos simultaneamente: o processo de projeto, mostrado na Figura 12.1, chegando ao projeto do banco de dados, e o projeto das aplicações, como uma atividade unificada. Alguns fornecedores, como Platinum, fornecem uma ferramenta para modelagem de dados e esquema de projeto (ERWin), e outra para modelagem de processo que gera o projeto funcional (BPWin). Outras ferramentas (como, por exemplo, o SECSI) usam a tecnologia de sistemas especialistas para guiar o processo de projeto, incluindo a perícia de projeto na forma de regras. A tecnologia de sistemas especialistas também é útil no levantamento e na análise de requisitos, que é normalmente um processo laborioso e frustrante. A tendência é usar tanto os repositórios de metadados quanto as ferramentas de projeto para melhorar os projetos de bancos de dados complexos. Sem pretensão de ser exaustiva, a Tabela 12.1 lista algumas das ferramentas mais conhecidas para o projeto de banco de dados e para a modelagem de aplicações. As empresas são relacionadas na tabela em ordem alfabética.

TABELA 12.1 Algumas das ferramentas de projeto de banco de dados automatizadas disponíveis atualmente

FUNCIONALIDADE**FERRAMENTA****COMPANHIA**

Embarcadero Technologies ER Studio

DB Artisan

Oracle

Popkin Software

Platinum Technology

Developer 2000 e Designer 2000

System Architect 2001

Platinum Enterprise Modeling Suite: ERWin, BPWin, Paradigm Plus

Persistence Inc. Powertier

Rational Rose Rational

Rogue Ware RW Metro

Resolution Ltd. XCase

Sybase Enterprise Ap]

Visio Visio Enterpri

Modelagem de banco de dados em ER e IDEFIX

Administração de banco de dados e gerenciamento de espaço e segurança

Modelagem de banco de dados, desenvolvimento de aplicações

Modelagem de dados, modelagem de objetos, modelagem de processos, análise/projeto estruturado

Modelagem de componentes de dados, processos e negócios

Mapeamento do modelo OO para modelo relacional

Modelagem UML e gerador de aplicações em C++ e em JAVA

Mapeamento do OO para o modelo relacional

Modelagem conceitual até a manutenção do código

Modelagem de dados, modelagem de lógica empresarial

Modelagem de dados, projeto e reengenharia em Visual Basic e Visual C++

12.6 RESUMO

Começamos este capítulo discutindo o papel de sistemas de informação em organizações; sistemas de banco de dados são vistos como uma parte dos sistemas de informação em grandes aplicações. Examinamos como os bancos de dados se ajustaram aos sistemas de informação para administração dos recursos de informação em uma organização e seu ciclo de vida. Vimos, então, as seis fases do processo de projeto. Geralmente, as três fases que compõem o projeto do banco de dados são o projeto conceitual, o projeto lógico (modelo de mapeamento de dados) e o projeto físico. Também discutimos a fase inicial de levantamento e análise de requisitos, que é considerada freqüentemente como uma *fase de anteprojeto*. Além disso, em algum ponto ao longo do projeto, deve ser escolhido um pacote SGBD específico. Discutimos alguns dos critérios organizacionais que entram em jogo para a seleção do SGBD. Como surgem problemas de desempenho e como são adicionadas novas aplicações, os projetos sofrem modificações. Foi destacada a importância de projetar tanto o esquema quanto as aplicações (ou transações).

Capítulo 12 Metodologia para Projeto Prático de Banco de Dados e Uso de Diagramas UML

Analizamos diferentes abordagens para o projeto do esquema conceitual e a diferença entre o projeto do esquema centralizado e da abordagem de integração de visões.

Apresentamos os diagramas da UML como um auxílio para a especificação de modelos de banco de dados e projeto; Introduzimos todo o conjunto de diagramas estruturais e de comportamento e, então, descrevemos o detalhamento notacional dos seguintes diagramas: caso de uso, sequência e estado. Já foram discutidos os diagramas de classe nas seções 3.8 e 4- (respectivamente).

Mostramos como são especificados os requisitos de um banco de dados universitário usando esses diagramas, e como eles podem ser usados para desenvolver o projeto conceitual do banco de dados. Foram fornecidos somente detalhes ilustrativos, e não a especificação completa. O Apêndice B, no site deste livro, desenvolve um estudo de caso completo do projeto e implementação de um banco de dados. Foram vistas, então, as atualmente famosas ferramentas para desenvolvimento de software — a Rational Rose e o Rose Data Modeler — que proporcionam suporte para o projeto conceitual para as fases de projeto lógico do banco de dados.

Rose é uma ferramenta muito mais ampla para o projeto completo de sistemas de informação.

Finalmente discutimos brevemente as características e as funcionalidades de ferramentas automatizadas comerciais para projeto de banco de dados, mais focadas no projeto do banco de dados que o Rose. Foi apresentado um resumo, em forma tabular, de suas características.

Questões para Revisão

- 12.1. Quais são as seis fases do projeto de um banco de dados? Discuta cada fase.
- 12.2. Quais das seis fases são consideradas, dentro do processo de projeto do banco de dados propriamente dito, atividades principais? Por quê?
- 12.3. Por que é importante projetar esquemas e aplicações em paralelo?
- 12.4. Por que é importante usar um modelo de dados independente da implementação durante o projeto de um esquema conceitual? Que modelos são usados em ferramentas de projeto atualmente? Por quê?
- 12.5. Discuta a importância do levantamento e da análise de requisitos.
- 12.6. Considere a aplicação atual de um sistema de banco de dados de seu interesse. Defina os requisitos dos diferentes níveis de usuários em termos dos dados necessários, tipos de consultas e processamento de transações.
- 12.7. Discuta as características que um modelo de dados deve possuir para o projeto de esquemas conceituais.
- 12.8. Compare e aponte os contrastes entre as duas principais abordagens de projeto de esquema conceitual.
- 12.9. Discuta as estratégias para o projeto de um único esquema conceitual a partir de seus requisitos.
- 12.10. Quais são os passos da abordagem de integração de visões para o projeto de um esquema conceitual? Quais são as dificuldades de cada um dos passos?
- 12.11. Como uma ferramenta de integração de visões trabalharia? Projete um exemplo de arquitetura modular para tal ferramenta.
- 12.12. Quais são as estratégias para integração de visões?
- 12.13. Discuta os fatores que influenciam na escolha de um SGBD para o sistema de informação de uma organização.
- 12.14. O que é mapeamento sistema-independente de um modelo de dados? Em que é diferente do mapeamento sistema-dependente de um modelo de dados?
- 12.15. Quais são os fatores que influenciam o projeto físico de banco de dados?
- 12.16. Discuta as decisões feitas durante um projeto físico de banco de dados.
- 12.17. Discuta os ciclos de vida macro e micro de um sistema de informação.
- 12.18. Discuta as diretrizes para o projeto físico de bancos de dados em SGBDRs.
- 12.19. Discuta os tipos de alterações que podem ser aplicadas ao projeto lógico de banco de dados de um banco de dados relacional.
- 12.20. Quais são as funções que as ferramentas de projeto de banco de dados normalmente proporcionam?
- 12.21. Que tipos de funcionalidades seriam desejáveis, nas ferramentas automatizadas, para apoiar projetos de grandes bancos de dados?

Bibliografia Seleccionada

Há uma vasta literatura em projeto de banco de dados. Relacionaremos, primeiro, alguns livros voltados ao projeto de banco de dados. Batini et al. (1992) fazem um amplo tratamento de projeto conceitual e lógico de banco de dados. Wiederhol (1986) cobre todas as fases de projeto de banco de dados, com ênfase no projeto físico. O'Neil (1994) apresenta uma discussão

12.6 Resumo 289

são detalhada do projeto físico e da emissão de transações em referência aos SGBDRs comerciais. Um grande conjunto de trabalhos sobre modelagem e projeto conceituai foi realizado nos anos 80. Brodie et al. (1984) fornecem um grupo de capítulos em modelagem conceituai, especificação de restrições e análise, bem como projeto de transações. Yao (1985) traz uma coleção de trabalhos que vão das técnicas de especificação de requisitos à reestruturação do esquema. Teorey (1998) enfatiza o modelo EER e discute vários aspectos do projeto conceituai e lógico do banco de dados. McFadden e Hoffer (1997) fazem uma boa introdução às metas das aplicações empresariais para a administração de banco de dados.

Navathe e Kerschberg (1986) discutem todas as fases do projeto de banco de dados e apontam o papel do dicionário de dados. Goldfine e König (1988) e ANSI (1989) discutem o papel do dicionário de dados no projeto de banco de dados. Rozen e Shasha (1991) e Carlis e March (1984) apresentam diferentes modelos para o problema do projeto físico do banco de dados. O projeto de banco de dados orientado a objetos é discutido em Schlaere Mellor (1988), Rumbaugh et al. (1991), Martin e Odell (1991) e Jacobson (1992). Livros recentes de Blahae Premerlani (1998) e Rumbaugh et al. (1999) consolidam as técnicas existentes em projeto orientado a objeto. Fowler e Scott (1997) fazem uma introdução rápida à UML.

Levantamento e análise de requisitos são tópicos pesadamente pesquisados. Chatzoglou et al. (1997) e Lubars et al. (1993) apresentam pesquisas sobre as práticas atuais de coleta de modelagem e análise de requisitos. Carroll (1995) proporciona um conjunto de leituras sobre o uso de cenários para levantamento de requisitos, capturados nos primeiros estágios do desenvolvimento de sistema. Wood e Silver (1989) fornecem uma boa avaliação oficial do processo de Joint Application Design (JAD). Potter et al. (1991) descrevem a notação e a metodologia Z para especificação formal de software. Zave (1997) classificou os esforços de pesquisa na engenharia de requisitos.

Um grande volume de trabalho foi produzido para os problemas de integração entre esquema e visão, que estão se tornando particularmente pertinentes hoje por causa da necessidade de integrar uma grande variedade de bancos de dados existentes. Navathe e Gadgil (1982) definem abordagens para a integração de visões. Metodologias para a integração de esquemas são comparadas em Batini et al. (1986). Um trabalho detalhado sobre a integração de visões n-árias pode ser encontrado em Navathe et al. (1986), Elmasri et al. (1986) e Larson et al. (1989). Uma ferramenta de integração, baseada em Elmasri et al. (1986), é descrita em Sheth et al. (1988). Outro sistema de integração de visão é discutido em Hayne e Ram (1990). Casanova et al. (1991) descrevem uma ferramenta para projeto modular de banco de dados. Motro (1987) discute a integração de bancos de dados preexistentes. A estratégia binária balanceada para integração de visões é discutida em Teorey e Fry (1982). Uma abordagem formal para integração de visões, que usa dependência de inclusão, é fornecida por Casanova e Vidal (1982). Ramesh e Ram (1997) descrevem uma metodologia para integração de relacionamentos em esquemas que utilizam o conhecimento das restrições de integridade; esse trabalho expande o trabalho prévio de Navathe et al. (1984a). Sheth et al. (1993) descrevem as metas para a construção do esquema global, argumentando sobre os relacionamentos entre atributos e equivalências de entidade. Navathe e Savasere (1996) descrevem uma abordagem prática para a construção do esquema global, baseada em operadores aplicados aos componentes do esquema. Santucci (1998) dá um tratamento detalhado de refinamento à integração de esquemas EER. Castano et al. (1999) apresenta uma pesquisa abrangente sobre as técnicas de análise de esquemas conceituais.

O projeto de transações é um tópico relativamente menos pesquisado. Mylopoulos et al. (1980) propuseram a linguagem TAXIS, e Albano et al. (1987) desenvolveram o sistema GALILEO, ambos abrangentes para a especificação de transações. A linguagem GORDAS, para o modelo ECR (Elmasri et al., 1985), contém capacidade para a especificação de transações. Navathe e Balaraman (1991) e Ngu (1991) discutem a modelagem de transação em geral para modelos de dados semânticos. Elmagarmid (1992) discute modelos de transações para aplicações avançadas. Batini et al. (1992, capítulos 8, 9, e 11) discutem o projeto de transações em nível alto e a análise de junções de dados e funções. Shasha (1992) é uma excelente fonte para sintonização de banco de dados. Podem ser encontradas informações sobre algumas ferramentas de projeto de banco de dados comerciais mais conhecidas nos Web sites dos vendedores (nomes das empresas na Tabela 12.1). Os princípios básicos, que suportam as ferramentas de projeto automatizadas, são discutidos em Batini et al. (1992, Capítulo 15). A ferramenta SECSI, francesa, é descrita em Metais et al. (1998). DKE (1997) é um assunto especial nas linguagens naturais de bancos de dados.

ggg

4

Armazenamento de Dados, Indexação, Processamento de Consultas e Projeto Físico

Os bancos de dados em geral são armazenados fisicamente como arquivos de registros em discos magnéticos. Este capítulo e o próximo tratam da organização de bancos de dados em estruturas de armazenamento e das técnicas para acessá-los eficientemente usando vários algoritmos, alguns dos quais requerem estruturas de dados auxiliares, chamadas índices. Começaremos pela Seção 13.1, apresentando os conceitos das hierarquias de armazenamento de computador e como elas são usadas em sistemas de bancos de dados. A Seção 13.2 é dedicada à descrição dos dispositivos de armazenamento em disco magnético e suas características, e também descreveremos brevemente os dispositivos de armazenamento em fita magnética. Tendo visto diferentes tecnologias de armazenamento, voltamos nossa atenção para os métodos de organização de dados em discos. A Seção 13.3 cobrirá a técnica de *buffering* duplo, utilizada para aumentar a velocidade de recuperação de múltiplos blocos de discos. Na Seção 13.4 veremos várias maneiras de formatar e armazenar registros em um arquivo no disco. A Seção 13.5 discutirá os vários tipos de operações que são geralmente realizadas com registros de um arquivo. A seguir apresentaremos três métodos primários para a organização de registros de um arquivo em disco: registros desordenados, analisados na Seção 13.6; registros ordenados na Seção 13.7; e registros *hashed* na Seção 13.8.

A Seção 13.9 discutirá, muito rapidamente, arquivos de registros mistos e outros métodos primários de organização de registros, tais como as árvores-B (B-trees). Essas estruturas são particularmente relevantes para o armazenamento de bancos de dados orientados a objetos, que serão vistos posteriormente nos capítulos 20 e 21. A Seção 13.10 descreve o RAID (*Redundant Arrays of Inexpensive [or Independent] Disks* — Séries Redundantes de Discos de Baixo Custo [ou Independentes]) —, uma arquitetura de sistema de armazenamento de dados que é freqüentemente usada em grandes empresas para melhor confiabilidade e desempenho. Finalmente, na Seção 13.11 descreveremos uma abordagem mais recente (as áreas de armazenamento em rede) para o gerenciamento de dados armazenados em redes. No Capítulo 14 veremos as técnicas para a criação de estruturas de dados auxiliares, chamadas índices, que aumentam a velocidade de pesquisa e de recuperação de registros. Tais técnicas envolvem o armazenamento de dados auxiliares, chamados arquivos de índices, além do próprio arquivo de registros.

Os capítulos 13 e 14 podem ser apenas folheados ou mesmo pulados por leitores que já tenham estudado organizações de arquivo. O conteúdo visto aqui será necessário para o entendimento dos capítulos 15 e 16, que tratam do processamento e da otimização de consultas.

13.1 INTRODUÇÃO

A coleção de dados que compõe um banco de dados computadorizado precisa ser armazenada fisicamente em alguma **mídia de armazenamento** de computador. Assim, os softwares SGBD podem recuperar, atualizar e processar esses dados conforme necessário. As mídias de armazenamento de computador formam uma *hierarquia de armazenamento* que inclui duas categorias principais:

- **Armazenamento primário.** Esta categoria inclui mídias de armazenamento que podem ser operadas diretamente pela *unidade central de processamento* (CPU), tais como a memória principal do computador e as menores, porém rápidas, memórias *cache*. O armazenamento primário geralmente fornece acesso rápido aos dados, mas sua capacidade é limitada.
- **Armazenamento secundário.** Esta categoria inclui discos magnéticos, discos ópticos e fitas. Esses dispositivos geralmente possuem maior capacidade, menor custo e proporcionam um acesso mais lento aos dados do que os dispositivos de armazenamento primário. Os dados de um armazenamento secundário não podem ser processados diretamente pela CPU; eles devem primeiro ser copiados em um armazenamento primário.

Primeiramente, na Seção 13.1.1, analisaremos como os bancos de dados são geralmente tratados na hierarquia de armazenamento e então, na Seção 13.1.2, apresentaremos uma visão geral dos vários dispositivos de armazenamento usados para armazenamento primário e secundário.

13.1.1 Hierarquias de Memórias e Dispositivos de Armazenamento

Em um sistema de computador moderno, os dados residem e são transportados por meio de uma hierarquia de mídias de armazenamento. A memória de maior velocidade é mais cara e, por isso, tem menor capacidade. A memória de menor velocidade como armazenamento em fita, que é *off-line*, tem capacidade de armazenamento potencialmente infinita.

No *nível de armazenamento primário*, a hierarquia de memória possui, na extremidade mais cara, a **memória *cache***, que é uma RAM (*Random Access Memory* — Memória de Acesso Aleatório) estática. A memória *cache* é geralmente usada pela CPU para acelerar a execução de programas. No próximo nível de armazenamento primário está a DRAM (RAM Dinâmica), que fornece a principal área de trabalho para a CPU manter seus programas e dados — é popularmente chamada de **memória principal**. A vantagem da DRAM é seu baixo custo, que continua a cair; a desvantagem é sua volatilidade e sua menor velocidade se comparada à RAM estática. No *nível de armazenamento secundário*, a hierarquia compreende discos magnéticos, bem como **armazenamento em massa** na forma de dispositivos de CD-ROM (*Compact Disk-Read-Only Memory* — Memória Apenas para Leitura em Disco Compacto), e finalmente as fitas, na extremidade mais barata da hierarquia. A **capacidade de armazenamento** é medida em kilobytes (Kbyte ou 1.000 bytes), megabytes (Mbyte ou 1 milhão de bytes), gigabytes (Gbyte ou 1 bilhão de bytes) e até mesmo em terabytes (1.000 Gbytes).

Os programas residem e são executados na DRAM. Geralmente os bancos de dados de grande porte que permanecem residentes em memória usam a memória secundária, e parcelas desse banco de dados são lidas e registradas em *buffers* da memória principal, conforme necessário. Agora que os computadores pessoais e as estações de trabalho possuem centenas de megabytes de dados em DRAM, está se tornando possível carregar uma grande fração de um banco de dados em memória principal. RAMs de 8 a 16 gigabytes em um único servidor estão se tornando comuns. Em alguns casos, bancos de dados inteiros podem ser mantidos na memória principal (com um *backup* em disco magnético), levando aos **bancos de dados em memória principal** — estes são particularmente úteis em aplicações de tempo real que exigem tempos de resposta extremamente rápidos. Um exemplo são as aplicações de comutação telefônica, que armazenam, em memória principal, os bancos de dados de roteamento e de linhas.

Entre o armazenamento em DRAM e em discos magnéticos, outra forma de memória, a **memória *flash***, está se tornando comum, particularmente porque é não-volátil. As memórias *flash* são de alta densidade e de alto desempenho, e usam a tecnologia EEPROM (*Electrically Erasable Programmable Read-Only Memory* — Memória Apenas de Leitura Eletricamente Programável). A vantagem da memória *flash* é sua alta velocidade de acesso; a desvantagem é que um bloco inteiro precisa ser apagado e escrito de cada vez. Cartões de memória *flash* estão aparecendo como mídia de armazenamento de dados em componentes com capacidades que variam de alguns megabytes a gigabytes. Eles vêm sendo usados em câmeras, aparelhos de MP3, acessórios de armazenamento USB etc. Os discos CD-ROM armazenam dados opticamente, que são lidos por um laser. Os CD-ROMs contêm dados pré-gravados que não podem ser sobrescritos. Os discos WORM (*Write'Once'Read'Memory* — Memória de Leitura de Única Gravação) são uma forma de armazenamento óptico usado para arquivar dados; eles permitem que os dados sejam escritos uma vez e lidos infinitas vezes, sem a possibilidade de serem apagados. Eles mantêm aproximadamente meio gigabyte de dados por disco e duram muito mais que os discos magnéticos. As **memórias ópticas *juke box*** usam uma série de discos CD-ROM, que são

1 Normalmente, as memórias voláteis perdem seu conteúdo no caso de falta de fonte de alimentação; o mesmo não ocorre com as memórias não-voláteis.

2 Por exemplo, o DD28F032SA da Intel é uma memória *flash* com capacidade de 32 megabytes, velocidade de acesso de 70 nanossegundos e taxa de transferência de escrita de 430 KB/segundo.

13.1 Introdução 295

carregados em unidades de leitura segundo a demanda. Embora as *juke box* ópticas tenham capacidade da ordem de centenas de gigabytes, seu tempo de recuperação é da ordem de centenas de milissegundos, muito mais lentos que os discos magnéticos. Esse tipo de armazenamento está em declínio em vista da rápida queda no custo e do aumento da capacidade dos discos magnéticos. O DVD (*Digital Video Disk* — Disco de Vídeo Digital) é um padrão recente dos discos ópticos, permitindo de 4,5 até 15 gigabytes de armazenamento por disco. A maioria das unidades de disco dos computadores pessoais atuais lê discos CD-ROM e DVD.

Finalmente, as **fitas magnéticas** são usadas para arquivo e *backup* de dados. As **fitas juke box** — que contêm um conjunto de fitas catalogadas e podem ser automaticamente gravadas em unidades de fitas — estão se tornando populares como **armazenamento terciário**, para manter terabytes de dados. Por exemplo, o sistema EOS (*Earth Observation Satellite* — Satélite de Observação da Terra) da NASA mantém bancos de dados arquivados dessa maneira.

Muitas grandes empresas já consideram normal manter bancos de dados da ordem de terabytes. O termo **banco de dados muito grande** (*very large database*) não pode mais ser definido com precisão, pois a capacidade de armazenamento em disco está em ascensão enquanto os custos estão em queda. Muito brevemente, o termo será reservado a bancos de dados que contenham dezenas de terabytes.

13.1.2 Armazenamento de Bancos de Dados

Os bancos de dados geralmente armazenam grandes quantidades de dados que devem ser *mantidos* por longos períodos de tempo. Os dados são acessados e processados repetidamente durante essa fase. Tal fato contrasta com a noção de estruturas de dados *transitórios* (*transient*), que são mantidos por tempo limitado durante a execução de um programa. Muitos bancos de dados são armazenados permanentemente (ou *persistentemente*) em armazenamento secundário de discos magnéticos pelas seguintes razões:

- Geralmente os bancos de dados são muito grandes para caberem inteiramente na memória principal.
- As circunstâncias que causam a perda permanente de dados armazenados aparecem menos freqüentemente em armazenamento secundário de disco do que em armazenamento primário. Assim, nos referimos a disco — e a outros dispositivos de armazenamento secundário — como **armazenamento não-volátil**, enquanto a memória principal é freqüentemente chamada **armazenamento volátil**.

- O custo de armazenamento por unidade de dado é uma ordem de magnitude menor para discos que para armazenamento primário. Algumas das tecnologias mais novas — tais como discos ópticos, DVDs e fitas *juke box* — provavelmente serão opções viáveis para o uso de discos magnéticos. Assim, no futuro, os bancos de dados poderão residir em níveis da hierarquia de memória diferentes daqueles descritos na Seção 13.1.1. Entretanto, pode-se antecipar que os discos magnéticos continuarão a ser a mídia de escolha primária para grandes bancos de dados nos próximos anos. Por isso, é importante estudar e compreender as propriedades e as características dos discos magnéticos e a maneira como os arquivos de dados podem ser neles organizados com o objetivo de projetar bancos de dados eficientes e com desempenho aceitável.

As fitas magnéticas são freqüentemente usadas como uma mídia de armazenamento para *backup* de bancos de dados porque o armazenamento em fita tem um custo ainda menor que o armazenamento em disco. Entretanto, o acesso a dados em fitas é mais lento. Os dados armazenados em fitas são *off-line*, isto é, é necessária alguma intervenção de um operador — ou de um dispositivo automático de carga — para carregar a fita antes que os dados se tornem disponíveis. Em contraste, os discos são dispositivos *on-line*, ou seja, podem ser acessados diretamente, a qualquer momento.

As técnicas utilizadas para armazenar grandes quantidades de dados estruturados em um disco são importantes para os projetistas de bancos de dados, bem como para o DBA e para os responsáveis pela implementação de SGBDs. Os projetistas de bancos de dados e o DBA devem conhecer as vantagens e as desvantagens de cada técnica de armazenamento quando projetam, implementam e operam um banco de dados em um SGBD específico. Geralmente o SGBD possui diversas opções disponíveis para a organização dos dados, e o processo de **projeto físico do banco de dados** envolve escolher, entre as opções disponíveis, as técnicas de organização que melhor se adaptem aos requisitos de uma dada aplicação. Os responsáveis pela implementação de sistemas SGBD devem estudar as técnicas de organização de dados de forma que possam implementá-las eficientemente e, assim, proporcionar opções suficientes ao DBA e aos usuários de SGBD.

Normalmente as aplicações comuns de bancos de dados necessitam de apenas uma pequena parte do banco de dados para processamento em um determinado momento. Quando certa porção dos dados for necessária, ela precisará ser localizada

- 3 Sua velocidade de rotação é mais lenta (aproximadamente 400 rpm), ocasionando maior período de latência e baixa taxa de transferência (aproximadamente 100 a 200 KB/segundo).

no disco, copiada para a memória principal para ser processada e, então, registrada, caso os dados tenham sido modificados. Os dados armazenados no disco são organizados como **arquivos de registros**. Cada registro é uma coleção de valores de dados que podem ser interpretados como fatos a respeito de entidades, bem como seus atributos e seus relacionamentos. Os registros devem ser armazenados no disco de maneira que seja possível localizá-los eficientemente quando necessário.

Há diversas **organizações primárias de arquivo** que determinam como os registros de um arquivo são *posicionados fisicamente* no disco e, portanto, *como eles podem ser acessados*. Um *arquivo heap* (ou *arquivo desordenado*) posiciona os registros no disco sem nenhuma ordem específica, por meio do acréscimo de novos registros ao final do arquivo; já um *arquivo ordenado* (*sorted* ou *arquivo sequencial*) mantém os registros ordenados segundo o valor de um campo em particular (chamado chave de ordenação). Um *arquivo hashed* usa uma função *hash* aplicada a um campo em particular (chamado chave de *hash*) para determinar a posição de um registro no disco. Outras organizações primárias de arquivo, tais como as *árvores-B* (B-trees), usam estruturas de árvores.

Discutiremos as organizações primárias de arquivo da Seção 13.6 até a Seção 13.9. Uma **organização secundária** ou uma **estrutura** de acesso **auxiliar** permite um acesso eficiente aos registros de um arquivo baseando-se em *campos alternativos* àqueles que tenham sido usados para a organização primária do arquivo. A maioria deles existe como índices e serão vistos no Capítulo 14.

13.2 DISPOSITIVOS DE ARMAZENAMENTO SECUNDÁRIO

Nesta seção descreveremos algumas características dos dispositivos de armazenamento em discos e fitas magnéticas. Os leitores que já tenham estudado esses dispositivos podem apenas folhear esta seção.

13.2.1 Descrição do Hardware dos Dispositivos de Disco

Os discos magnéticos são utilizados para armazenamento de grandes quantidades de dados. A unidade mais básica de dados em um disco é um único **bit** de informação. Por meio da magnetização de uma área do disco pode-se representar o valor de um bit como 0 (zero) ou 1 (um). Para codificar a informação, os bits são agrupados em **bytes** (ou **caracteres**). Geralmente um byte tem de quatro a oito bits, dependendo do computador e do dispositivo. Consideraremos que um caractere é armazenado em um único byte e usaremos os termos *byte* e *caractere* como sinônimos. A **capacidade** de um disco é o número de bytes que ele pode armazenar, e esse número é geralmente muito grande. Os pequenos discos flexíveis usados em microcomputadores têm capacidade entre 400 Kbytes e 1,5 Mbytes; discos rígidos para micros geralmente mantêm entre diversas centenas de Mbytes até alguns poucos Gbytes; e um grande conjunto de discos usado em servidores e mainframes tem capacidade que chega a algumas poucas dezenas ou centenas de Gbytes. A capacidade dos discos continua crescendo conforme avança a tecnologia.

Seja qual for sua capacidade, todos os discos são feitos de material magnético moldado como um fino disco circular (Figura 13.1a) e protegido por uma cobertura plástica ou acrílica. Um disco é **face única** se ele armazenar informações em apenas uma de suas superfícies e **dupla face** se ambas as superfícies forem utilizadas. Para aumentar a capacidade de armazenamento, os discos são montados em um **conjunto (pacote) de discos** (Figura 13.1b), que pode compreender muitos discos e, assim, muitas superfícies. A informação é armazenada na superfície do disco em círculos concêntricos de *pequenas espessuras*, cada qual possuindo um diâmetro distinto. Cada círculo é chamado de **trilha**. Para os conjuntos de discos, as trilhas de mesmo diâmetro entre as várias superfícies compõem um **cilindro** por causa do formato que produziriam caso se conectassem no espaço. O conceito de cilindro é importante porque os dados armazenados em um cilindro podem ser recuperados muito mais rapidamente do que se fossem distribuídos em diferentes cilindros.

O número de trilhas em um disco varia de algumas centenas a milhares de trilhas, e a capacidade de cada trilha geralmente varia de dezenas de Kbytes até 150 Kbytes. Como uma trilha geralmente contém uma grande quantidade de informação, ela é dividida em blocos ou setores menores. A divisão da trilha em **setores** é codificada na superfície de um disco e não pode ser alterada. Um tipo de organização por setor chama de setor uma parte da trilha determinada por um ângulo fixo a partir do centro (Figura 13.2a). Diversas outras organizações por setor são possíveis; em uma delas, os setores são determinados por ângulos menores a partir do centro conforme se move para fora, assim a densidade de gravação é mantida uniforme (Figura 13.2b). Uma técnica chamada ZBR (*Zone Bit Recording*—Gravação por Zona de Bit) permite que uma faixa de cilindros tenha o mesmo número de setores por arco. Por exemplo, os cilindros 0-99 podem ter um setor por trilha, os cilindros 100-199 podem ter dois setores por trilha etc. Nem todos os discos têm suas trilhas divididas em setores.

4 Atualmente, em alguns discos, os círculos são conectados em um tipo de espiral contínua.

13.2 Dispositivos de Armazenamento Secundário 297

trilha

(a)

(b)

rotação do disco

cilindro de

trilhas (imaginário)

movimento do atuador

FIGURA 13.1 (a) Um disco de face única com hardware de leitura/escrita, (b) Um conjunto de discos com hardware de leitura/escrita.

(a) Trilha

Setor (arco de uma trilha)

(b)

Três setores Dois setores Um setor

FIGURA 13.2 Diferentes organizações de setor em disco, (a) Setores formados por um ângulo fixo. (b) Setores que mantêm uma densidade uniforme de gravação.

A divisão de uma trilha em **blocos de discos (ou páginas)** de mesmo tamanho é feita pelo sistema operacional durante a **formatação (ou inicialização)** do disco. O tamanho do bloco é fixado durante a inicialização e não pode ser alterado dinamicamente. O tamanho de bloco de disco geralmente varia entre 512 e 4096 bytes. Um disco com setores codificados freqüentemente tem os setores subdivididos em blocos durante a inicialização. Os blocos são separados pelos *interblock gaps*

298 Capítulo 13 Armazenamento em Disco, Estruturas Básicas de Arquivos e *Hashing*

(**intervalo entre blocos**) de tamanho fixo, que incluem informações de controle especialmente codificadas gravadas durante a inicialização do disco. Essa informação é usada para determinar qual bloco na trilha segue cada *interblock gap*. A Tabela 13.1 apresenta as especificações de um disco típico.

TABELA 13.1 Especificações dos discos topo de linha Cheetah da Seagate.

Descrição

Número do modelo

Formato (largura)

Altura

Largura

Peso

Cheetah X15 36LP

ST336732LC 3,5 polegadas 25,4 mm 101,6 mm 0,68 kg

Cheetah 10K.6

ST3146807LC 3,5 polegadas 25,4 mm 101,6 mm

0,73 kg

Capacidade/Interface

Capacidade formatado Tipo de interface

36,7 Gbytes 80 pinos

146,8 Gbytes 80 pinos

Configuração

Número de discos (físico) Número de cabeçotes (físico) Número de cilindros Bytes por setor Densidade de área Densidade de trilha

Densidade de gravação

18.479 512 n.d. n.d

n.d

4

8

49.854

512

36.000 Mbits/polegada²

64-000 trilhas/polegada

570.000 bits/polegada

Desempenho

Taxas de transferência

Taxa de transferência interna (min.) 522 Mbits/seg

Taxa de transferência interna (máx.) 709 Mbits/seg

Taxa de transferência int. formatado (min.) 51 Mbytes/seg

Taxa de transferência int. formatado (máx.) 69 Mbytes/seg

Taxa de transferência entrada e saída externa 320 Mbytes/seg

475 Mbits/seg 840 Mbits/seg 43 Mbytes/seg 78 Mbytes/seg 320 Mbytes/seg

Tempo de pesquisa

Tempo de busca médio (leitura) Tempo de busca médio (escrita) Busca trilha a trilha, leitura Busca trilha a trilha, escrita Latência média

Outro

Tamanho padrão de buffer (cache) Velocidade de rotação

3,6 mseg (típica) 4,2 mseg (típica) 0,5 mseg (típica) 0,8 mseg (típica) 2 mseg

8.192 Kbytes T5Krpm

4,7 mseg (típica) 5,2 mseg (típica) 0,3 mseg (típica) 0,5 mseg (típica) 2,99 mseg

8.000 Kbytes 10Krpm

13.2 Dispositivos de Armazenamento Secundário 299

TABELA 13.1 Especificações dos discos topo de linha Cheetah da Seagate, (continuação)

Descrição	Cheetah X15 36LP	Cheetah 10K.6 Confiabilidade
Tempo médio entre falhas (MTBF)	1.200.000 horas	1.200.000 horas
Erros de leitura recuperáveis	10 por 10 ¹ bits	10 por 10 ¹ bits
Erros de leitura não-recuperáveis	1 por 10 ¹ bits	1 por 10 ¹ bits
Erros de busca	10 por 10 bits	10 por 10 bits

(Cortesia da Seagate Technology)

Há uma melhoria contínua na capacidade de armazenamento e nas taxas de transferência associadas aos discos; eles também estão progressivamente mais baratos — atualmente, 1 megabyte de armazenamento em disco custa apenas uma fração de dólar. Os custos estão caindo tão rapidamente que 0,1 centavo de dólar/MB, que significa 1 dólar/GB e 1.000 dólares/TB não está muito longe de acontecer.

Um disco é um dispositivo endereçável de *acesso aleatório*. A transferência de dados entre a memória principal e o disco se dá em unidades de blocos de disco. O endereço físico de um bloco — uma combinação do número do cilindro, do número da trilha (número da superfície dentro do cilindro no qual a trilha está localizada) e do número do bloco (dentro da trilha) — é fornecido ao hardware de entrada e saída do disco. Em muitas unidades de disco modernas, um único número, chamado LBA (*Logical Block Address* — Endereço Lógico de Bloco), que está entre 0 e n (supondo que a capacidade total do disco seja de $n+1$ blocos), é mapeado automaticamente para o bloco correto pela controladora da unidade de disco. O endereço de um **buffer** — uma área contígua reservada no armazenamento principal que mantém um bloco — também é fornecido. Em um comando de **leitura**, o bloco é copiado do disco para o buffer; já no comando de **escrita** (gravação), o conteúdo do buffer é copiado para o bloco do disco. Algumas vezes diversos blocos contíguos, chamados de **cluster**, podem ser transferidos como uma unidade. Nesse caso, o tamanho do buffer é ajustado para corresponder ao número de bytes no cluster.

O mecanismo de hardware que realmente lê ou escreve um bloco é o **cabeçote de leitura/escrita** do disco, que é parte de um sistema chamado **unidade de disco**. Um disco ou um conjunto de discos é montado na unidade de disco, que inclui um motor que rotaciona os discos. Um cabeçote de leitura/escrita inclui um componente eletrônico vinculado a um **braço mecânico**. Os conjuntos de discos com múltiplas superfícies são controlados por diversos cabeçotes de leitura/escrita — um para cada superfície (Figura 13.1b). Todos os braços são conectados a um **atuador** vinculado a outro motor elétrico, que move os cabeçotes de leitura/escrita simultaneamente e os posiciona com precisão sobre o cilindro das trilhas especificadas em um endereço de um bloco.

As unidades de disco de discos rígidos rotacionam o conjunto de discos continuamente a uma velocidade constante (geralmente entre 5.400 e 15.000 rpm). Para um disco flexível, a unidade de disco começa a rotacionar o disco sempre que um pedido específico de leitura ou escrita for iniciado e cessa a rotação assim que a transferência de dados se completa. Tão logo o cabeçote de leitura/escrita esteja posicionado na trilha correta, o bloco especificado no endereço de bloco se move sob o cabeçote de leitura/escrita e o componente eletrônico do cabeçote é ativado para transferir os dados. Algumas unidades de disco possuem cabeçotes de leitura/escrita fixos, tantas quantas forem as trilhas. São os chamados **discos com cabeçote fixo**, enquanto as unidades de disco com um atuador são chamadas **discos com cabeçote móvel**. Nos discos com cabeçote fixo, uma trilha ou um cilindro é selecionado por meio de um dispositivo da seleção eletrônica do cabeçote de leitura/escrita apropriado em vez dos movimentos mecânicos reais; conseqüentemente, isso é muito mais rápido. Entretanto, o custo de cabeçotes de leitura/escrita adicionais é um tanto alto, por isso, discos de cabeçote fixo não são usados com frequência.

Uma **controladora de disco**, geralmente embutida em uma unidade de disco, controla a unidade de disco e realiza a *interface* entre ela e o sistema do computador. Um dos padrões de *interface* utilizados hoje em dia para unidades de disco em PCs e estações de trabalho é chamado SCSI (*Small Computer Storage Interface* — Interface de Armazenamento de Computadores Pequenos). A controladora aceita comandos de entrada e saída de alto nível e executa a ação apropriada para posicionar o braço e fazer com que a ação de leitura/escrita ocorra. Para transferir um bloco de disco, dado o seu endereço, a controladora de disco deve primeiro posicionar mecanicamente o cabeçote de leitura/escrita sobre a trilha correta. O tempo necessário para fazê-lo é chamado de **tempo da pesquisa**. Tempos normais de pesquisa vão de 7 a 10 milissegundos em computadores pessoais e de 3 a 8 milissegundos em servidores. Seguindo este, há outro atraso — chamado **atraso rotacional** ou **latência** —, até que o início do bloco desejado rotacione até a posição sob o cabeçote de leitura/escrita. Esse atraso depende da quantidade de rotações por minuto do disco. Por exemplo, a 15.000 rpm, o tempo de cada rotação é de 4 milissegundos, e a média de atraso rotacional é a metade do tempo de uma rotação, ou seja, 2 milissegundos. Finalmente, algum tempo adicional é necessário para a transferência dos dados; esse tempo é chamado de **tempo de transferência do bloco**. Assim, o tempo total necessário para localizar e transferir um bloco arbitrário, dado o seu endereço, é a soma do tempo da pesquisa, do atraso rotacional e do tempo

de transferência do bloco. O tempo de pesquisa e o atraso rotacional são geralmente muito maiores que o tempo da transferência do bloco. Para realizar a transferência de múltiplos blocos com maior eficiência, é comum a transferência de diversos blocos consecutivos da mesma trilha ou do mesmo cilindro. Isso elimina o tempo de pesquisa e o atraso rotacional para todos os blocos, exceto do primeiro, e pode resultar em uma economia de tempo substancial quando vários blocos contíguos forem transferidos. Geralmente, o fabricante do disco informa uma **taxa de transferência de volume** para calcular o tempo necessário para transferência de blocos consecutivos. O Apêndice B, no site deste livro, traz uma discussão sobre esse e outros parâmetros de disco. O tempo necessário para localizar e transferir um bloco de disco é da ordem de milissegundos, geralmente variando entre 12 e 60 milissegundos. Para blocos contíguos, localizar o primeiro bloco leva de 12 a 60 milissegundos, mas a transferência dos blocos subsequentes pode levar apenas de 1 a 2 milissegundos cada. Diversas técnicas de pesquisa tiram vantagem da recuperação consecutiva de blocos quando pesquisam dados no disco. Em qualquer caso, o tempo de transferência da ordem de milissegundos é considerado bastante alto se comparado ao tempo necessário para processar dados na memória principal por meio das CPUs atuais. Portanto, localizar dados no disco é o *maior gargalo* em aplicações de bancos de dados. As estruturas de arquivo que discutimos aqui e no Capítulo 14 tentam *minimizar o número de transferências de blocos* necessários para localizar e transferir os dados do disco para a memória principal.

13.2.2 Dispositivos de Armazenamento em Fitas Magnéticas

Os discos são dispositivos de armazenamento secundário de **acesso aleatório** porque um bloco de disco arbitrário pode ser acessado 'aleatoriamente', desde que especifiquemos seu endereço. As fitas magnéticas são dispositivos de **acesso seqüencial**; para acessar o *n-ésimo* bloco na fita, devemos primeiro percorrer os *n-1* blocos precedentes. Os dados são armazenados em cartuchos de fitas magnéticas de alta capacidade, similares às fitas de áudio ou vídeo. É necessária uma **unidade de fita** para ler os dados ou para gravá-los no **cartucho de fita**. Em geral, cada grupo de bits que formam um byte é armazenado de maneira transversal na fita e, por sua vez, os bytes são armazenados consecutivamente no carretel.

Um cabeçote de leitura/escrita é usado para ler ou gravar dados na fita. Os registros de dados na fita também são armazenados em blocos — embora os blocos possam ser substancialmente maiores que os dos discos e os *interblock gaps* (intervalos entre blocos) também sejam bem maiores. Como as densidades de fitas típicas são de 1.600 a 6.250 bytes por polegada, um intervalo entre blocos normal de 0,6 polegada corresponde a uma faixa entre 960 e 3.750 bytes de espaço de armazenamento desperdiçado. Para melhor utilização do espaço, é comum agrupar muitos registros em um bloco.

A principal característica de uma fita é o acesso aos blocos de dados na **ordem seqüencial**. Para acessar um bloco no meio de um cartucho, a fita é montada e, então, percorrida até que o bloco requisitado esteja sob o cabeçote de leitura/escrita. Por essa razão o acesso à fita pode ser lento e as fitas podem não ser utilizadas para armazenar dados on-line, exceto para aplicações específicas. Entretanto, as fitas atendem a uma função muito importante: o **backup** do banco de dados. Uma razão para fazer o backup é manter cópias dos arquivos em disco para o caso de os dados serem perdidos por causa de um desastre de disco (*disk crash*), que pode ocorrer se o cabeçote de leitura/escrita tocar a superfície do disco por conta de algum mau funcionamento mecânico. Por essa razão, os arquivos em disco são periodicamente copiados em fita. Para muitas aplicações críticas *on-line*, tais como sistemas de reservas aéreas, a fim de evitar qualquer interrupção, sistemas de espelhamento são usados para manter três conjuntos de discos idênticos — dois operando *on-line* e um como *backup*. Aqui, os discos *off-line* tornam-se um dispositivo de *backup*. Os três discos são rotacionados de forma que possam ser trocados no caso de haver uma falha em uma das unidades de disco ativas. As fitas também podem ser usadas para armazenar arquivos de bancos de dados demasiadamente grandes. Finalmente, arquivos de bancos de dados que são raramente utilizados ou que estão desatualizados, embora sejam necessários para manter registros históricos, podem ser **arquivados** em fita. Recentemente, fitas magnéticas menores, de 8 mm (similares às usadas em filmadoras), que podem armazenar até 50 Gbytes, bem como cartuchos de fita *helical scan* (varredura helicoidal) e CDs e DVDs graváveis, têm se tornado mídias populares para o *backup* de arquivos de dados de estações de trabalho e computadores pessoais. Também são usados para armazenar imagens e bibliotecas de sistema. Fazer o *backup* de bancos de dados corporativos, de forma que nenhuma informação de transação se perca, é uma importante incumbência. Atualmente, bibliotecas de fitas com posições para diversas centenas de cartuchos são usadas com fitas *Digital Linear* e *Super-digital Linear* (DLTs e SLTs — Fita Linear Digital e Fita Linear Superdigital), com capacidade de centenas de gigabytes, que gravam dados em trilhas lineares. Braços robóticos são utilizados para escrever paralelamente em cartuchos múltiplos, usando várias unidades de fitas, com softwares de rotulagem automática para identificar os cartuchos de backup. Um exemplo de biblioteca gigante é o modelo L5500 da Storage Technology, que pode ser configurado até 13,2 petabytes (1 petabyte = 1.000 TB), com uma vazão (throughput) de 55TB/hora. Deixaremos a discussão sobre a tecnologia de armazenamento em disco, chamada RAID, e sobre a área de armazenamento em rede para o final do capítulo.

5 Chamado *interrecord gap* (intervalo entre registros) na terminologia de fitas.

13.3 BUFFERING DE BLOCOS

Quando é necessário transferir diversos blocos do disco para a memória principal, e todos os endereços de blocos são conhecidos, diversos *buffers* podem ser reservados na memória principal para acelerar a transferência. Enquanto um *buffer* estiver sendo lido ou escrito, a CPU pode processar os dados em outro *buffer*. Isso é possível porque existe um processador (controlador) de entrada e saída independente que, uma vez inicializado, pode executar a transferência de um bloco de dados entre a memória e o disco de maneira independente e simultaneamente ao processamento da CPU.

A Figura 13.3 ilustra como dois processos podem ser conduzidos em paralelo. Os processos A e B estão sendo executados concorrentemente, de **modo** intercalado, enquanto os processos C e D estão sendo executados **concorrentemente, de modo paralelo**. Quando uma única CPU controla múltiplos processos, a execução em paralelo não é possível. Entretanto, os processos ainda podem ser executados concorrentemente de forma intercalada. O *buffering* é muito útil para processos executados concorrentemente em paralelo, ou porque está disponível um processador exclusivo para entrada e saída de disco, ou porque existem diversos processadores.

Concorrência intercalada das operações A e B.

Execução em paralelo das operações C e D.

Tempo

FIGURA 13.3 Execução em concorrência intercalada *versus* paralela.

A Figura 13.4 ilustra como leitura e processamento podem ser conduzidos em paralelo quando o tempo necessário para processar um bloco de disco na memória for menor que o tempo necessário para ler o próximo bloco e preencher um *buffer*. A CPU pode iniciar o processamento do bloco após sua transferência para a memória principal. Ao mesmo tempo, o processador de entrada e saída do disco pode ler e transferir o próximo bloco para um *buffer* diferente. Essa técnica é chamada de **buffering duplo** e também pode ser usada para gravar um fluxo contínuo de blocos, da memória para o disco. O *buffering* duplo permite leitura ou escrita contínua de dados em blocos consecutivos de disco, o que elimina o tempo de pesquisa e o atraso rotacional para transferência de todos os blocos, exceto o primeiro. Além disso, os dados estão sempre prontos para o processamento, reduzindo, assim, o tempo de espera nos programas.

Bloco do disco Entrada e saída

Bloco do disco Processamento

```

i          i + 1
Preencher A , Preencher B          i + 2          i + 3 Preencher A
Preencher B          i + 4 Preencher A
|          i
i Processar A          i+1          |          i+2 Processar B          i          Processar A          i + 3
Processar B          i + 4 Processar A
i

```

FIGURA 13.4 Uso de dois *buffers*, A e B, para ler o disco.

13.4 DISPOSIÇÃO DE REGISTROS DE ARQUIVOS EM DISCO

Nesta seção definiremos os conceitos de registros, tipos de registros e arquivos. A seguir analisaremos as técnicas para posicionar registros de arquivos no disco.

13.4.1 Registros e Tipos de Registros

Os dados geralmente são armazenados na forma de **registros**. Cada registro consiste de uma coleção de **valores** ou **itens** relacionados, na qual cada valor é formado por um ou mais bytes e corresponde a um **campo** de dado registro. Os registros geralmente descrevem as entidades e seus atributos. Por exemplo, um registro EMPREGADO representa uma entidade empregado e cada valor de campo do registro especifica algum atributo daquele empregado, tal como NOME, DATANASC, SALÁRIO ou SUPERVISOR. Uma coleção de nomes de campo e seus respectivos tipos de dados constituem uma definição de **tipo de registro** ou **formato de registro**. Um **tipo de dado**, associado a cada campo, especifica os tipos de valores que um campo pode receber. O tipo de dado de um campo geralmente é um dos tipos de dados padrões usados em programação. Eles incluem tipos numéricos (inteiro, inteiro longo ou ponto flutuante), cadeia de caracteres (de tamanho fixo ou variável), booleano (tendo apenas 0 e 1 ou VERDADEIRO e FALSO como valores) e, algumas vezes, tipos de dados especialmente codificados para **data/hora**. O número de bytes necessário para cada tipo de dado é fixo para um determinado sistema de computador. Um número inteiro pode necessitar de 4 bytes, um inteiro longo, de 8 bytes, um número real, de 4 bytes, um booleano, de 1 byte, uma data de 10 bytes (supondo o formato AAAA-MM-DD), e uma cadeia de caracteres de tamanho fixo, de k bytes. Cadeias de caracteres de tamanho variável podem necessitar de tantos bytes quantos forem os caracteres em cada valor de campo. Por exemplo, o tipo do registro EMPREGADO pode ser definido — usando a notação da linguagem de programação C — de acordo com a seguinte estrutura:

```
struct empregado{
char nome[30];
char ssn[9];
int salário;
int código;
char departamento[20];
};
```

Em aplicações recentes de bancos de dados, pode haver necessidade de armazenar itens de dados que consistem de grandes objetos desestruturados, que representam imagens, vídeo ou áudio digitalizado, ou texto em formato livre. Estes são referidos como BLOBS (*Binary Large Objects* — Grandes Objetos Binários). Um item de dado BLOB geralmente é armazenado separadamente do seu registro em um conjunto de blocos de discos, e um ponteiro para o BLOB é incluído no registro.

13.4.2 Arquivos, Registros de Tamanho Fixo e Registros de Tamanho Variável

Um **arquivo** é uma *seqüência* de registros. Em muitos casos, todos os registros de um arquivo são do mesmo tipo de registro. Se todos os registros em um arquivo possuem exatamente o mesmo tamanho (em bytes), diz-se que o arquivo é formado por **registros de tamanho fixo**. Se registros diferentes em um arquivo possuem tamanhos diferentes, diz-se que o arquivo é formado por **registros de tamanho variável**. Um arquivo pode ter registros de tamanho variável por diversas razões:

- Os registros do arquivo são do mesmo tipo de registro, mas um ou mais campos são de tamanho variável (**campos de tamanho variável**). Por exemplo, o campo NOME de EMPREGADO pode ser um campo de tamanho variável.
- Os registros do arquivo são do mesmo tipo de registro, porém, um ou mais campos podem ter diversos valores para registros individuais; tal campo é chamado **campo multivalorado** (*repeating field*), e um grupo de valores para o campo é chamado **grupo de repetição**.
- Os registros do arquivo são do mesmo tipo de registro, mas um ou mais campos são **opcionais**, ou seja, eles podem ter valores para alguns, mas não para todos os registros do arquivo (**campos opcionais**).
- O arquivo contém registros de *diferentes tipos de registro* e, portanto, de tamanhos variáveis (**arquivo misto**). Isso pode ocorrer se registros relacionados de diferentes tipos forem *reunidos* (em *cluster*) nos blocos de disco; por exemplo, OS registros do BOLETIM de um determinado estudante podem ser posicionados seguindo o seu registro ALUNO.

Os registros de EMPREGADO de tamanho fixo na Figura 13.5a possuem um tamanho de registro de 71 bytes. Cada registro possui os mesmos campos, e os tamanhos dos campos são fixos, de forma que o sistema possa identificar a posição do byte inicial

13.4 Disposição de Registros de Arquivos em Disco 303

de cada campo a partir da posição inicial do registro. Isso facilita a localização dos valores dos campos pelos programas que acessam tais arquivos. Observe que é possível representar um arquivo que logicamente deveria ter registros de tamanho variável como um arquivo de registros de tamanho fixo. Por exemplo, no caso de campos opcionais, poderíamos ter *cada campo* incluído em *cada registro do arquivo*, porém, um valor especial nulo seria armazenado se nenhum valor existisse para aquele campo. Para um campo multivalorado, poderíamos alocar tantos espaços em cada registro quanto o *número máximo de valores* que o registro pode possuir. Em qualquer um dos casos, haverá desperdício de espaço se certos registros não possuírem valores para todos os espaços físicos fornecidos em cada registro. Agora consideremos outras opções de formatação de registros de um arquivo de registros de tamanho variável.

Para *campos de tamanho variável*, cada registro possui um valor para cada campo, entretanto, não sabemos o tamanho exato de alguns dos valores desses campos. Para determinar os bytes de um registro em particular, podemos usar caracteres **separadores** especiais (tais como ? ou % ou \$) — que não aparecerem em nenhum valor de campo — para encerrar os campos de valor variável (Figura 13.5.b); ou ainda podemos armazenar o tamanho, em bytes, do campo no registro, antes do valor do campo.

(a) NOME

CÓDIGO SSN SALÁRIO / DEPARTAMENTO DATA_CONTR

1

31

40 44 48

i

68

(c)

CÓDIGO
SALÁRIO

(b) NOME SSN

Smith, John | 123456789

12

DEPARTAMENTO

ICU

separadores

~~Z ~ I Caracteres

Computador

21 25 29

NOME=Smith, John SSN=123456789 | DEPARTAMENTO=Computador jfl

Caracteres separadores

= Separa nome de campo de valor de campo

I Separa

campos

Finaliza registro

FIGURA 13.5 Três formatos de armazenamento de registro, (a) Registro de tamanho fixo com seis campos e tamanho de 71 bytes, (b) Um registro com dois campos de tamanho variável e três campos de tamanho fixo. (c) Um registro de tamanho variável com três tipos de caracteres separadores.

Um arquivo de registros, com *campos opcionais*, pode ser formatado de diferentes maneiras. Se o número total de campos para um tipo de registro for grande, mas o número de campos que realmente aparece em um registro típico for pequeno, podemos incluir em cada registro uma sequência de pares <nome-de-campo, valor-de-campo> em vez de apenas valores de campos. Três tipos de caracteres separadores são utilizados na Figura 13.7c, embora possamos usar o mesmo caractere separador para os dois primeiros casos — separar o nome do campo do valor do campo e um campo do próximo campo. Uma opção mais prática seria designar um código curto de **tipo de campo** — digamos, um número inteiro — para cada campo e incluir em cada registro uma sequência de pares <tipo-de-campo, valor-de-campo> em vez de pares <nome-de-campo, valor-de-campo>.

Um *campo multivalorado* precisaria de um caractere para separar os valores repetidos de um campo e de outro caractere separador para indicar o término do campo. Finalmente, para um arquivo que inclua *registros de diferentes tipos*, cada registro é precedido por um indicador de **tipo de registro**. Compreensivelmente, programas que processam arquivos de registros de tamanho variável — que geralmente são parte do sistema de arquivo e, portanto, pouco visíveis aos programadores comuns —

304 Capítulo 13 Armazenamento em Disco, Estruturas Básicas de Arquivos e *Hashing*

precisam ser mais complexos que aqueles para registros de tamanho fixo, nos quais a posição inicial e o tamanho de cada campo são conhecidos e fixos.

13.4.3 Divisão de Registros em Blocos e Registros *Spanned Versus* Registros *Não-Spanned (Unspanned)*

Os registros de um arquivo precisam ser alocados em blocos de discos porque um bloco é a *unidade de transferência de dados* entre o disco e a memória. Quando o tamanho do bloco é maior que o tamanho do registro, cada bloco conterá vários registros, embora raramente alguns arquivos possam ter registros tão extensos que não caibam em um bloco. Suponha que o tamanho do bloco é B bytes. Para um arquivo de registros de tamanho fixo de R bytes, com $B > R$, podemos colocar $bfr = LB/RJ$ registros por bloco, onde $\lfloor x \rfloor$ (*função arredondamento para baixo*) arredonda para baixo o número x para um valor inteiro. O valor de bfr é chamado **fator blocagem (fator de divisão em blocos)** para o arquivo. Em geral, R pode não dividir B exatamente; assim, em cada bloco, devemos ter espaço não utilizado igual a

$B - (bfr * R)$ bytes

Para usar esse espaço sem uso, podemos armazenar parte de um registro em um bloco e o restante em outro. Ao final do primeiro bloco, um **ponteiro** aponta para o bloco que contém o restante do registro no caso de ele não ser um bloco consecutivo no disco. Essa organização é chamada ***spanned*** porque os registros podem se fragmentar por mais de um bloco. Sempre que um registro for maior que um bloco, *devemos* usar uma organização *spanned*. Se os registros não puderem atravessar as fronteiras dos blocos, a organização é chamada de ***não-spanned***. Essa organização é usada com registros de tamanho fixo, tendo $B > R$, porque isso faz com que cada registro comece em uma localização conhecida no bloco, simplificando o processamento dos registros. Para registros de tamanho variável, podem ser usadas tanto a organização *spanned* quanto a não-*spanned*. Se o tamanho médio dos registros é grande, é vantajoso usar a *spanned* para reduzir a perda de espaço em cada bloco. A Figura 13.6 ilustra comparativamente as organizações *spanned* e não-*spanned*.

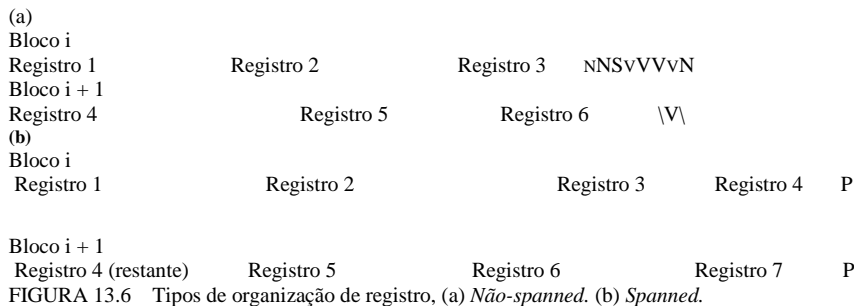


FIGURA 13.6 Tipos de organização de registro, (a) *Não-spanned*. (b) *Spanned*.

Para registros de tamanho variável que utilizam a organização *spanned*, cada bloco pode armazenar um número diferente de registros. Nesse caso, o fator blocagem bfr representa o número *médio* de registros por blocos para o arquivo. Podemos usar o bfr para calcular o número de blocos b necessários para um arquivo de r registros:

$b = \lfloor (rlbfr) \rfloor$ blocos

onde $\lceil x \rceil$ (*função arredondamento para cima*) arredonda o valor de x para o próximo inteiro.

13.4.4 Alocação de Blocos de Arquivos em Disco

Há diversas técnicas padronizadas para a alocação de blocos de um arquivo em disco. Na **alocação consecutiva** (*contiguous allocation*), os blocos de arquivo são alocados consecutivamente no disco. Utilizando-se o *buffering* duplo, a leitura do arquivo inteiro é muito rápida, porém, torna difícil a expansão do arquivo. Na alocação **encadeada** (*linked allocation*), cada bloco

6 Outros esquemas são possíveis para a representação de registros de tamanho variável.

13.5 Operações em Arquivos 305

de arquivo contém um ponteiro para o próximo bloco de arquivo. Isso facilita a expansão do arquivo, mas torna lenta a leitura do arquivo inteiro. Uma combinação dos dois seria a alocação de blocos consecutivos em *clusters* de disco com o encadeamento desses *clusters*. Os *clusters* são, às vezes, chamados segmentos de **arquivo** ou extensões. Outra possibilidade é utilizar a alocação indexada, na qual um ou mais blocos de índice contém ponteiros para os blocos de arquivo de fato. Também é comum utilizar combinações dessas técnicas.

13.4.5 Cabeçalhos de Arquivo

Um **cabeçalho de arquivo** ou **descriptor de arquivo** contém informações sobre o arquivo que serão necessárias aos programas de sistema que acessam os registros do arquivo. O cabeçalho contém informações para determinar os endereços de disco dos blocos do arquivo, bem como registrar descrições do formato que podem incluir tamanho e ordem dos campos em um registro para registros de tamanho fixo *não-spanned*, além de códigos de tipos de campos, caracteres separadores e códigos de tipos de registro para registros de tamanho variável.

Para pesquisar um registro no disco, um ou mais blocos são copiados nos *buffers* da memória principal. Os programas, então, procuram o registro ou os registros desejados dentro dos *buffers* usando a informação do cabeçalho do arquivo. Se o endereço do bloco que contém o registro desejado não é conhecido, os programas de pesquisa devem fazer uma **pesquisa linear** pelos blocos do arquivo. Cada bloco de arquivo é copiado em *buffer* e pesquisado até o registro ser localizado, ou até que todos os blocos do arquivo tenham sido pesquisados sem sucesso. Para um arquivo grande, isso pode consumir muito tempo. O objetivo de uma boa organização de arquivo é localizar um bloco que contenha um registro desejado com um número mínimo de transferências de bloco.

13.5 OPERAÇÕES EM ARQUIVOS

As operações em arquivo são geralmente divididas em operações **de recuperação** e em operações **de atualização**. As primeiras não alteram nenhum valor no arquivo, apenas localizam certos registros, de forma que seus valores de campo possam ser examinados e processados. As últimas mudam o arquivo por meio da inclusão ou da exclusão de registros ou pela modificação de valores dos campos. Em ambos os casos, podemos ter de **selecionar** um ou mais registros para a recuperação, a exclusão ou a modificação, tendo como base uma condição **de seleção** (ou condição **para filtragem**), que especifica os critérios que o registro ou os registros desejados devem satisfazer.

Considere um arquivo EMPREGADO com os campos NOME, SSN, SALÁRIO, CODIGOCARGO e DEPARTAMENTO. Uma **condição de seleção simples** pode envolver uma comparação de igualdade sobre algum valor de campo — por exemplo, (SSN = '123456789') ou (DEPARTAMENTO = 'Pesquisa'). Condições mais complexas podem envolver outros tipos de operadores de comparação, tais como < ou >; um exemplo é (SALÁRIO > 30.000). O caso geral é ter uma expressão booleana arbitrária sobre os campos como condição de seleção.

As operações de seleção em arquivos geralmente são baseadas em condições de seleção simples. Uma condição complexa precisa ser decomposta pelo SGBD (ou pelo programador) para extrair uma condição simples que possa ser usada para localizar registros no arquivo. Cada registro localizado é, então, verificado para determinar se ele satisfaz a condição de seleção completa. Por exemplo, podemos extrair a condição simples (DEPARTAMENTO = 'Pesquisa') da condição complexa ((SALÁRIO > 30.000) AND (DEPARTAMENTO = 'Pesquisa')); cada registro que satisfaça (DEPARTAMENTO = 'Pesquisa') será localizado e testado para verificar se também satisfaz (SALÁRIO > 30.000).

Quando diversos registros satisfizerem uma condição de pesquisa, o *primeiro* registro — relativo à sequência física dos registros do arquivo — é inicialmente localizado e designado como **registro atual**. Operações de pesquisa subsequentes começam a partir desse registro e localizam o *próximo* registro no arquivo que satisfaça a condição.

Operações reais para a localização e o acesso de registros de arquivos variam de sistema para sistema. Abaixo, apresentamos um conjunto representativo de operações. Em geral, programas de alto nível, como programas de software SGBD, são usados para acesso aos registros usando esses comandos, assim, algumas vezes nos referiremos a **variáveis de programas** com as seguintes descrições:

- **Open** (Abrir): Prepara o arquivo para leitura ou escrita. Aloca os *buffers* adequados (pelo menos dois) para hospedar os blocos do arquivo desde o disco, e busca o cabeçalho do arquivo. Posiciona o ponteiro do arquivo no início do arquivo.
- **Reset** (Reinicializar): Reposiciona o ponteiro de arquivo, de um arquivo aberto, em seu início.

- **Find** ou **Locate** (Encontrar ou Localizar): Busca o primeiro registro que satisfaça uma condição de pesquisa. Transfere o bloco que contém aquele registro para um *buffer* na memória principal (se ele ainda não estiver lá). O ponteiro do arquivo aponta para o registro no *buffer* e ele se torna o *registro atual*. Algumas vezes, verbos diferentes são utilizados para indicar se o registro localizado deve ser recuperado ou atualizado.
- **Read** ou **Get** (Ler ou Obter): Copia o registro atual do *buffer* para uma variável do programa do usuário. Este comando também pode avançar o ponteiro do registro corrente para o próximo registro do arquivo, o que pode implicar a leitura no disco do próximo bloco.
- **FindNext** (Encontrar o Próximo): Procura o próximo registro no arquivo que satisfaça a condição de pesquisa. Transfere o bloco contendo aquele registro para um *buffer* da memória principal (se ele já não estiver lá). O registro é localizado no *buffer* e se torna o registro atual.
- **Delete** (Excluir): Exclui o registro atual e (no final) atualiza o arquivo no disco para refletir a exclusão.
- **Modify** (Modificar): Modifica alguns valores de campos no registro atual e (no final) atualiza o arquivo no disco para refletir a modificação.
- **Insert** (Incluir): Acrescenta um novo registro no arquivo por meio da localização do bloco no qual o registro deve ser incluído, transferindo aquele bloco para o *buffer* da memória principal (se ele já não estiver lá), escrevendo o registro no *buffer* e (no final) escrevendo o *buffer* no disco para refletir a inclusão.
- **Close** (Fechar): Finaliza o acesso ao arquivo por meio da liberação dos *buffers* e da execução de quaisquer outras operações de limpeza necessárias.

As operações anteriores (exceto *Open* e *Close*) são chamadas **record-at-a-time** (um registro por vez), porque cada operação se aplica a um único registro. É possível agrupar as operações *Find*, *FindNext* e *Read* em uma única operação *Scan* (Varrer), cuja descrição é a seguinte:

- **Scan** (Varrer): Se o arquivo tiver acabado de ser aberto ou reinicializado, o *Scan* retorna o primeiro registro; caso contrário, retorna o próximo. Se uma condição for especificada com a operação, o registro retornado é o primeiro ou o próximo registro que satisfizer a condição.

Em sistemas de bancos de dados, operações adicionais de mais alto nível, **set-at-a-time** (um conjunto de registros por vez), podem ser aplicadas a um arquivo. Exemplos dessas operações:

- **FindAll** (Encontrar Todos): Encontra todos os registros de um arquivo que satisfizerem uma condição de pesquisa.
- **Find** ou **Locate n** (Encontrar ou Localizar n): Busca o primeiro registro que satisfizer a condição de pesquisa e, então, continua localizando os próximos *n-1* registros que satisfizerem a mesma condição. Transfere os blocos contendo os *n* registros para o *buffer* da memória principal (se já não estiverem lá).
- **FindOrdered** (Encontrar e Ordenar): Recupera todos os registros de um arquivo em alguma ordem especificada.
- **Reorganize** (Reorganizar): Inicia o processo de reorganização. Como veremos, algumas organizações de arquivo necessitam de reorganização periódica. Um exemplo é reordenar os registros do arquivo, classificando-os de acordo com um campo específico. Nesse ponto, vale a pena observar a diferença entre os termos *organização de arquivo* e *método de acesso*. Uma **organização de arquivo** se refere à organização dos dados de um arquivo em registros, blocos e estruturas de acesso; e isso inclui a maneira como registros e blocos são posicionados e interligados na mídia de armazenamento. Um **método de acesso**, porém, fornece um grupo de operações — como as listadas anteriormente — que podem ser aplicadas a um arquivo. Em geral, é possível aplicar vários métodos de acesso a uma organização de arquivo. No entanto, alguns métodos de acesso podem ser aplicados apenas a arquivos organizados de certa maneira. Por exemplo, não podemos aplicar um método de acesso indexado a um arquivo sem índices (Capítulo 6). Geralmente temos a expectativa de usar algumas condições de pesquisa mais do que outras. Alguns arquivos podem ser estáticos, significando que operações de atualização são raramente executadas; outros arquivos, mais **dinâmicos**, podem ser alterados freqüentemente, assim, operações de atualização são aplicadas constantemente a eles. Uma organização bem-sucedida de arquivo deve realizar, tão eficientemente quanto possível, as operações que temos expectativa de que sejam *aplicadas freqüentemente* a ele. Por exemplo, considere o arquivo EMPREGADO (Figura 13.5a), que armazena os registros dos atuais empregados de uma empresa. Esperamos acrescentar registros (quando empregados são contratados), excluir registros (quando empregados deixam a empresa) e modificar registros (por exemplo, quando o salário ou o cargo de um funcionário mudar). A exclusão ou modificação de um registro requer uma condição de seleção para identificar um registro em particular ou um conjunto de registros. A recuperação de um ou mais registros também requer uma condição de seleção.

13.6 Arquivos de Registros Desordenados (*Heap Files*) 307

Se os usuários esperarem principalmente aplicar condições de pesquisa baseadas no SSN, o projetista deve escolher uma organização de arquivo que facilite a localização de um registro dado seu valor para o campo SSN. Isso pode implicar a ordenação fisicamente dos registros pelo valor de SSN ou a definição de um índice para o SSN (Capítulo 6). Suponha que uma segunda aplicação use o arquivo para gerar os cheques de pagamento dos empregados e necessite que eles sejam agrupados por departamento. Para tal aplicação, é melhor armazenar consecutivamente todos os registros de empregados que possuam o mesmo valor para departamento, reunindo-os em blocos e, talvez, ordenando-os pelo nome dentro de cada departamento. Entretanto, esse arranjo entra em conflito com a ordem dos registros segundo o valor do SSN. Se ambas aplicações forem importantes, o projetista deveria escolher uma organização que permitisse que elas fossem executadas de maneira eficiente. Infelizmente, em muitos casos, pode não existir uma organização que permita que todas as operações necessárias em um arquivo sejam implementadas eficientemente. Em tais casos, deve ser escolhida uma solução conciliatória que leve em consideração a importância e o elenco das operações de recuperação e de atualização esperadas.

Nas seções seguintes e no Capítulo 6, discutiremos métodos para a organização dos registros de um arquivo em disco. Várias técnicas genéricas, tais como classificação, *hashing* e indexação, são usadas para criar métodos de acesso. Além disso, várias técnicas genéricas para a manipulação de inclusões e exclusões funcionam com muitas organizações de arquivo.

13.6 ARQUIVOS DE REGISTROS DESORDENADOS (*HEAP FILES*)

No tipo de organização mais simples e básica, os registros estão posicionados no arquivo segundo a ordem pela qual foram incluídos, de forma que novos registros são acrescentados ao final do arquivo. Tal organização é chamada *heap file* ou *pile file* (arquivo pilha). Essa organização é freqüentemente utilizada com caminhos de acesso adicionais, tais como os índices secundários vistos no Capítulo 6. Ela também é usada para coletar e armazenar registros para uso futuro.

Incluir um novo registro é *muito eficiente*: o último bloco de disco do arquivo é copiado no *buffer*, o novo registro é acrescentado e o bloco é reescrito no disco. O endereço do último bloco do arquivo é mantido no cabeçalho do arquivo. Entretanto, a pesquisa por um registro, usando qualquer condição, envolve uma **pesquisa seqüencial** bloco a bloco do arquivo — um procedimento dispendioso. Se apenas um dos registros satisfizer a condição de pesquisa, então um programa irá carregar em memória e pesquisar, na média, a metade dos blocos do arquivo antes de encontrar o registro desejado. Para um arquivo com b blocos, isso implica pesquisar $(b/2)$ blocos, em média. Se nenhum registro, ou vários deles, satisfizer a condição de pesquisa, o programa deve ler e pesquisar todos os b blocos do arquivo.

Para excluir um registro, um programa deve primeiro encontrar seu bloco, copiá-lo em um *buffer*, então excluir o registro do *buffer* e, finalmente, **reescrever o bloco** de volta no disco. Isso deixa espaços sem uso no bloco do disco. A exclusão de um grande número de registros resulta, dessa forma, em desperdício de espaço de armazenamento. Outra técnica utilizada para a exclusão de registro é possuir um byte ou bit extra, chamado **marcador de exclusão** e armazenado com cada registro. Um registro é excluído por meio do ajuste do marcador de exclusão para um certo valor. Um valor diferente para o marcador indica um registro válido (não excluído). Programas de pesquisa consideram apenas os registros válidos de um bloco quando executam sua busca. Ambas as técnicas de exclusão exigem a **reorganização** periódica do arquivo para reaproveitar o espaço, sem uso, dos registros excluídos. Durante a reorganização, os blocos do arquivo são acessados consecutivamente, e os registros são agrupados por meio da remoção dos registros excluídos. Após tal reorganização, os blocos são, uma vez mais, preenchidos até sua capacidade. Outra possibilidade é utilizar o espaço dos registros excluídos durante a inclusão de novos registros, embora isso exija controles adicionais para manter-se dados sobre as localizações vazias.

Podemos usar ambas as organizações, *spanned* e *não-spanned*, para arquivos desordenados, e elas podem ser usadas tanto com registros de tamanho fixo quanto com registros de tamanho variável. A modificação de um registro de tamanho variável pode exigir a exclusão do registro antigo e a inclusão de um registro modificado, porque o registro modificado pode não caber em seu espaço antigo no disco.

Para ler todos os registros pela ordem dos valores de algum campo, criamos uma cópia ordenada do arquivo. Ordenar (ou classificar) é uma operação dispendiosa em grandes arquivos de disco, e técnicas especiais para **classificação externa** (*external sort*) são usadas no Capítulo 15.

Para um arquivo desordenado com *registros de tamanho fixo* utilizando *blocos não-spanned* e *alocação consecutiva*, é fácil acessar qualquer registro pela sua posição no arquivo. Se os registros do arquivo são numerados por $0, 1, 2, \dots, r-1$ e os registros em cada bloco são numerados por $0, 1, 2, \dots, bfr-1$, onde bfr é o fator de blocagem, então o i -ésimo registro do arquivo está localizado no bloco $\lfloor (ibfr) \rfloor$ e é o $(i \bmod bfr)$ -ésimo registro daquele bloco. Tal arquivo é freqüentemente chamado de **arquivo relativo** ou **direto** porque os registros podem ser acessados diretamente pelas suas posições relativas. O acesso a um registro pela sua

7 Às vezes essa organização é chamada arquivo seqüencial.

308 Capítulo 13 Armazenamento em Disco, Estruturas Básicas de Arquivos e *Hashing*

posição não ajuda a localizar um registro com base em uma condição de pesquisa; entretanto, **facilita a** construção de caminhos de acesso ao arquivo, tais como os índices vistos no Capítulo 6.

13.7 ARQUIVOS DE REGISTROS ORDENADOS (*SORTED FILES*)

Podemos ordenar fisicamente os registros de um arquivo em um disco a partir dos valores de um de seus campos — chamado **campo de classificação**. Isso leva a um arquivo **ordenado** ou **seqüencial**. Se o campo de classificação também for um **campo chave** do arquivo — um campo do qual se garante ter um único valor em cada registro —, então o campo é chamado **chave de classificação** para o arquivo. A Figura 13.7 mostra um arquivo ordenado, tendo o campo NOME como campo-chave de classificação (supondo que os empregados tenham nomes distintos).

Arquivos ordenados têm algumas vantagens sobre arquivos desordenados. Primeiro, a leitura dos registros na ordem de valores da chave de classificação se torna extremamente eficiente porque nenhuma classificação se faz necessária. Segundo, encontrar o próximo registro a partir do atual, na ordem dos valores da chave de classificação, geralmente não requer acesso a outros blocos porque o próximo registro deve estar no mesmo bloco que o atual (a menos que o registro corrente seja o último do bloco). Terceiro, o uso de uma condição de pesquisa baseada no valor do campo-chave de classificação resulta em um acesso mais rápido quando a técnica de pesquisa binária é utilizada, a qual constitui uma melhoria em relação à pesquisa linear, embora não seja freqüentemente utilizada em arquivos de disco.

Uma **pesquisa binária** em arquivos de disco pode ser feita com base em blocos em vez de em registros. Suponha que um arquivo possui b blocos numerados por $1, 2, \dots, b$; que os registros são ordenados pelo valor crescente de seu campo-chave de classificação; e que estamos buscando um registro cujo valor do campo-chave de classificação é K . Supondo que os endereços de disco dos blocos do arquivo estão disponíveis no cabeçalho do arquivo, a pesquisa binária pode ser descrita pelo Algoritmo 13.1. Uma pesquisa binária geralmente acessa $\log_2(b)$ blocos, quer o registro seja encontrado ou não — uma melhoria em relação às pesquisas lineares, nas quais, na média, $(b/2)$ blocos são acessados quando o registro for encontrado e b blocos são acessados quando o registro não for encontrado.

Algoritmo 13.1: Pesquisa binária baseada em uma chave de classificação de um arquivo em disco.

```

 $I \leftarrow 1$ ;  $u \leftarrow b$ ; (*  $b$  é o número de blocos do arquivo *) while ( $u \geq I$ ) do
begin
 $i \leftarrow (I + u) \div 2$ ; read bloco  $i$  do arquivo para o buffer;
if  $K < (\text{valor do campo-chave de classificação do primeiro registro do bloco } i)$  then  $u \leftarrow i - 1$ 
else if  $K > (\text{valor do campo-chave de classificação do último registro do bloco } i)$  then  $I \leftarrow i + 1$ 
else if registro com valor do campo-chave de classificação =  $K$  esta no buffer then goto encontrado else goto naoencontrado; end;
goto naoencontrado;
```

Um critério de pesquisa envolvendo as condições $>$, $<$, $>=$ e $<=$ para o campo de classificação será bastante eficiente, uma vez que a ordem física dos registros implica que todos os registros que satisfaçam a condição serão consecutivos no arquivo. Por exemplo, referindo-se à Figura 13.7, se o critério de pesquisa for (NOME $<$ 'G') — onde $<$ significa *alfabeticamente anterior* —, os registros que satisfizerem o critério de pesquisa serão aqueles a partir do início do arquivo até o primeiro registro que possuir o valor de NOME começando com a letra G.

Classificar não traz nenhuma vantagem em relação ao acesso aleatório ou ordenado aos registros com base em valor dos demais *campos não ordenados* do arquivo. Nesses casos, executaremos uma pesquisa linear para o acesso aleatório. Para acessar os registros na ordem de um campo não ordenado, é necessário criar uma outra cópia classificada do arquivo — em uma ordem diferente.

8 O termo *arquivo seqüencial* também tem sido utilizado para se referir a arquivos desordenados.

13.7 Arquivos de Registros Ordenados (*Sorted Files*) 309

bloco 1

NOME

SSN DATANASC CARGO SALÁRIO SEXO

Aaron, Ed

Abbott, Diane

•

Acosta, Marc

bloco 2

Adams, John

Adams, Robin

/

Akers, Jan

bloco 3

Alexander, Ed

Alfred, Bob

;

Allen, Sam

bloco 4

Allen, Troy

Anders, Keith

Anderson, Rob

bloco 5

Anderson, Zach

Angeli, Joe

•

Archer, Sue

bloco 6

Arnold, Mack

Arnold, Steven

;

Atkins, Timothy

bloco n -1

Wong, James

Wood, Donald

•

Woods, Manny

bloco n

Wright, Pam

Wyatt, Charles

•

Zimmer, Byron

FIGURA 13.7 Alguns blocos de um arquivo ordenado (seqüencial) de registros de EMPREGADO tendo NOME como campo-chave de classificação.

A inclusão e a exclusão de registros são operações dispendiosas para um arquivo ordenado porque os registros deverão permanecer ordenados fisicamente. Para inserir um registro, devemos encontrar sua posição correta no arquivo, de acordo com seu valor no campo de classificação e, então, abrir espaço no arquivo para inserir o registro naquela posição. Para um arquivo grande, isso pode consumir muito tempo, pois, em média, a metade dos registros do arquivo deverá ser transferida para abrir espaço para o novo registro. Isso significa que a metade dos blocos de arquivos deverá ser lida e regravada após os registros serem transferidos entre eles. Para a exclusão de um registro, o problema será menos grave se forem utilizados marcadores de exclusão e se forem feitas reorganizações periódicas.

Uma opção para tornar a inclusão mais eficiente é manter alguns espaços não utilizados em cada bloco para novos registros. Entretanto, uma vez que esse espaço seja utilizado, o problema original ressurge. Um outro método frequentemente utilizado é criar um arquivo *desordenado* temporário, chamado arquivo de *overflow* ou de transação. Com essa técnica, o arquivo ordenado real é chamado arquivo principal ou mestre. Os novos registros são incluídos ao final do arquivo de *overflow* em vez de serem incluídos em suas posições corretas no arquivo principal. Periodicamente o arquivo de *overflow* é classificado e incorporado ao arquivo mestre durante a reorganização do arquivo. A inclusão se torna muito eficiente, porém, à custa do aumento da complexidade do algoritmo de pesquisa. O arquivo de *overflow* deve ser pesquisado por meio de uma pesquisa linear se, após a pesquisa binária, o registro não for encontrado no arquivo principal. Para aplicações que não exijam informação atualizada, os registros de *overflow* poderão ser ignorados durante a busca.

Modificar o valor de um campo de registro depende de dois fatores: (1) da condição para localizar o registro e (2) do campo a ser modificado. Se a condição de pesquisa envolver o campo-chave ordenado, poderemos localizar o registro usando uma pesquisa binária; caso contrário, precisaremos realizar uma pesquisa linear. Um campo não ordenado poderá ser modificado por meio da alteração do registro e de sua regravação na mesma localização física no disco — supondo registros de tamanho fixo. Modificar um campo de classificação significa que sua posição poderá ser alterada no arquivo, o que exigirá a exclusão do registro antigo seguida da inclusão do registro modificado.

A leitura dos registros do arquivo pela ordem do campo de classificação é bastante eficiente se ignorarmos registros de *overflow*, uma vez que os blocos podem ser lidos consecutivamente usando *buffering* duplo. Para incluir os registros em *overflow*, deveremos proceder à sua intercalação nas posições corretas; nesse caso, poderíamos primeiro reorganizar o arquivo e então ler seus blocos seqüencialmente. Para reorganizar o arquivo, primeiro precisamos classificar os registros no arquivo de *overflow* e, então, mesclá-los ao arquivo-mestre. Os registros marcados para a exclusão serão removidos durante a reorganização.

A Tabela 13.2 resume o tempo médio de acesso em acessos de bloco para encontrar um registro específico em um arquivo com b blocos.

TABELA 13.2 Tempos médios de acesso para organizações básicas de arquivo

Ípo de Organização	Método de Acesso/resquisa	Tempo Médio de Acesso a um
Registro em Particular		
<i>Heap</i> (Desordenado)	Varredura Seqüencial (Pesquisa Linear)	$b/2$
Ordenado	Varredura Seqüencial	$b/2$
Ordenado	Pesquisa Binária	$\log_2 b$

Raramente os arquivos ordenados são utilizados em aplicações de bancos de dados, a menos que um caminho de acesso real, chamado índice primário, seja utilizado; isso resulta em um arquivo seqüencial indexado e melhora ainda mais o tempo de acesso aleatório no campo-chave de classificação. Analisaremos índices no Capítulo 14.

13.8 TÉCNICAS DE HASHING

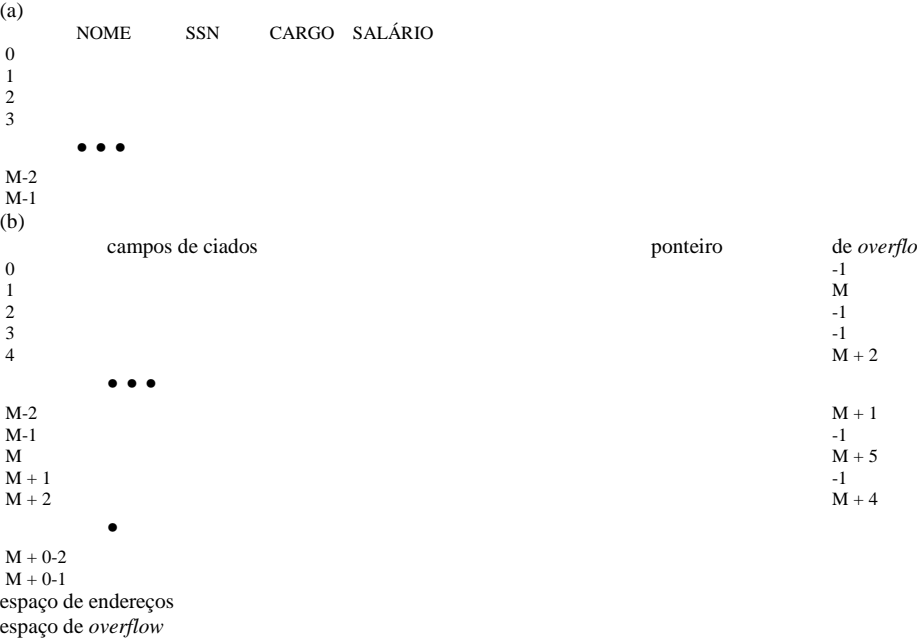
Um outro tipo de organização primária de arquivo baseia-se em *hashing*, que fornece um acesso muito rápido aos registros sob certas condições de pesquisa. Essa organização é geralmente chamada arquivo *hash*. A condição de pesquisa precisa ser de igualdade em um único campo; neste caso, o campo é chamado campo de hash. Na maioria dos casos, o campo de *hash* será também o campo-chave do arquivo e, assim, será chamado chave de *hash*. A idéia do hashing é fornecer uma função h , chamada função hash ou função *randomizing*, que, aplicada ao valor do campo de hash de um registro, gere o *endereço* do bloco de disco no qual o registro está armazenado. A busca do registro dentro do bloco pode ser realizada em um *buffer* da memória principal. Para a maioria dos registros, necessitamos de apenas um bloco para recuperar esse registro.

O *hashing* também é utilizado como uma estrutura de pesquisa interna em um programa sempre que um grupo de registros for acessado exclusivamente pelo valor de um campo. Descreveremos o uso de *hashing* para arquivos internos na Seção 13.9.1; depois mostraremos como ele pode ser modificado para armazenar arquivos externos em disco na Seção 13.9.2. Na Seção 13.9.3 discutiremos as técnicas para estender o *hashing* a arquivos com crescimento dinâmico.

9 Um arquivo *hash* também é chamado *arquivo direto*.

13.8.1 Hashing Interno

Para arquivos internos, o *hashing* é normalmente implementado por uma **tabela hash**, por meio do uso de um vetor de registros. Suponha que o índice do vetor varie de 0 a M-1 (Figura 13.8a); temos, assim, M posições (**slots**), cujos endereços correspondem aos índices do vetor. Escolheremos uma função *hash* tal que transforme o valor do campo de *hash* em um número inteiro entre 0 e M-1. Uma função *hash* típica é a função $h(K) = K \bmod M$, que retorna o valor inteiro do resto após a divisão de uma chave *hash* K por M; esse valor será, então, usado como o endereço do registro.



1

- ponteiro *null* = -1.
- o ponteiro de *overflow* se refere à posição do próximo registro na lista encadeada

FIGURA 13.8 Estruturas de dados de *hashing* interno, (a) Vetor de M posições para uso em *hashing* interno, (b) Resolução de colisão por meio do encadeamento de registros.

Valores de campos de *hash* não inteiros podem ser transformados em inteiros antes que a função *mod* seja aplicada. Para cadeias de caracteres, o código numérico (ASCII), associado aos caracteres, pode ser utilizado na transformação — por exemplo, multiplicando seus valores no código. Para um campo de *hash* cujo tipo de dado é uma cadeia de vinte caracteres, o Algoritmo 13.2a pode ser usado para calcular o endereço *hash*. Supusemos que a função *code* retorna o código numérico de um caractere e que nos é dado um valor K de campo de *hash* do tipo K: *array[1..20] of char* (em PASCAL) ou *char K[20]* (em C).

Algoritmo 13.2: Dois algoritmos simples de *hashing*. (a) Aplicação da função *hash mod* a uma cadeia de caracteres K. (b) Resolução de colisão por meio de *open addressing*.

```
(a) temp <- 1;
for i <- 1 to 20 do temp <- temp endereco_hash <- temp mod M;
code(K[i]) mod «;
```

```

(b)  $i \leftarrow \text{endereco\_hash}(K)$ ;  $a \leftarrow i$ ; if posição  $i$  esta ocupada
then begin  $i \leftarrow (i + 1) \bmod W$ ;
while ( $i \neq a$ ) and posição  $i$  esta ocupada
do  $i \leftarrow (i + 1) \bmod M$ ; if ( $i = a$ ) then todas as posições estão preenchidas
else  $\text{novo\_endereco\_hash} \leftarrow i$ ; end;

```

Outras funções *hash* podem ser usadas. Uma técnica, chamada **folding**, envolve a aplicação de uma função aritmética tal como a *adição*, ou de uma função lógica, como exclusivo, a partes diferentes do valor do campo de *hash* para calcular o endereço *hash*. Uma outra técnica envolve escolher alguns dígitos do valor do campo de *hash* — por exemplo, o terceiro, o quinto e o oitavo dígitos — para formar o endereço *hash*. O problema com a maioria das funções *hash* é que elas não garantem que diferentes valores levarão a diferentes endereços *hash* porque o **espaço de variação do campo de hash** — o número de valores possíveis que um campo de *hash* pode assumir — é geralmente muito maior que o **espaço de endereços** — o número de endereços disponíveis para os registros. A função *hash* mapeia o espaço de variação do campo de *hash* no espaço de endereço.

Uma **colisão** ocorrerá quando o valor do campo de *hash* de um registro que está sendo incluído levar a um endereço que já contiver um registro diferente. Nessa situação, deveremos incluir o novo registro em alguma outra posição, uma que o endereço *hash* não está ocupado. O processo para encontrar outra posição é chamado **resolução de colisão**. Há vários métodos para a resolução de colisão, vejamos alguns:

- **Open addressing (endereço aberto)**: A partir da posição já ocupada pelo endereço *hash*, o programa prossegue a verificação, pela ordem, das posições subsequentes, até que seja encontrada uma posição não utilizada (vazia). O Algoritmo 13.2b pode ser usado com esse propósito.

- **Encadeamento (chaining)**: Neste método são mantidas várias posições de *overflow*, geralmente por meio da exteri do vetor por um número de posições de *overflow*. Além disso, um campo ponteiro é adicionado a cada localização do registro. Uma colisão é resolvida posicionando o novo registro em uma localização de *overflow* não utilizada e achando o endereço de *overflow* no ponteiro do endereço *hash* ocupado. Assim, será mantida uma lista encadeada de registros de *overflow* para cada endereço *hash*, conforme mostrado na Figura 13.8b.

- **Hashing múltiplo**: O programa aplicará uma segunda função *hash* caso a primeira resulte em colisão. Se novamente ocorrer uma colisão, o programa usará *open addressing* ou aplicará uma terceira função *hash*, usando *open addressing* necessário. Cada método de resolução de colisão requer seus próprios algoritmos para inclusão, recuperação e exclusão de registros. Os algoritmos de encadeamento são os mais simples, porém, os algoritmos de exclusão para *open addressing* são mais complicados. Livros básicos sobre estruturas de dados discutem algoritmos de *hash* com mais detalhes.

O objetivo de uma boa função *hash* é distribuir os registros uniformemente pelo espaço de endereços de forma a minimizar as colisões e não deixar endereços sem uso. Estudos de simulação e análise mostraram que geralmente é melhor manter entre 70% e 90% uma tabela *hash* cheia, tal que o número de colisões permaneça pequeno e não haja muito desperdício de espaço. Desse modo, se nossa expectativa é de que tenhamos r registros para armazenar em uma tabela, devemos escolher localizações para o espaço de endereços, de modo que (r/M) esteja entre 0,7 e 0,9. Também pode ser útil escolher um número primo para M , uma vez que foi demonstrado que isso melhora a distribuição dos endereços *hash* pelo espaço de endereço quando a função *hash mod* é utilizada. Outras funções *hash* podem exigir que M seja uma potência de 2.

13.8.2 Hashing Externo para Arquivos em Disco

Hashing para arquivos em disco são chamados **hashing externos**. Para adaptar-se às características do armazenamento em disco, considera-se que o espaço de endereço alvo é constituído de **buckets (baldes)**, cada qual mantendo múltiplos registros. Um *bucket* é um bloco de disco ou um grupo (*cluster*) de blocos consecutivos. A função *hash* mapeia uma chave a um número de *bucket* relativo em vez de atribuir um endereço absoluto de bloco para o *bucket*. Uma tabela, mantida no cabeçalho do arquivo, converte o número do *bucket* para o endereço de bloco de disco correspondente, conforme é ilustrado na Figura 13.9.

10 Uma discussão detalhada sobre as funções *hash* está fora do objetivo de nossa apresentação.

13.8 Técnicas de *Hashing* 313

endereço
 número do bloco
 de *bucket* no disco
 0
 1 2
 M 2 M-1

FIGURA 13.9 Correspondência entre números de *bucket* e endereços de blocos de disco.

O problema da colisão é menos grave com *buckets* porque poderão ser direcionados pela função *hash* a um mesmo *bucket*, tantos quantos forem os registros que nele couberem, sem causar problemas. Entretanto, devemos nos preparar para o caso de um *bucket* atingir sua capacidade, e um novo registro, que esteja sendo incluído, seja levado pela função *hash* para aquele *bucket*. Podemos utilizar uma variação do encadeamento no qual seja mantido, em cada *bucket*, um ponteiro para uma lista encadeada de registros de *overflow* do *bucket*, conforme mostrado na Figura 13.10. Os ponteiros na lista encadeada devem ser **ponteiros de registros**, que incluem tanto um endereço de bloco quanto a posição relativa do registro no bloco.

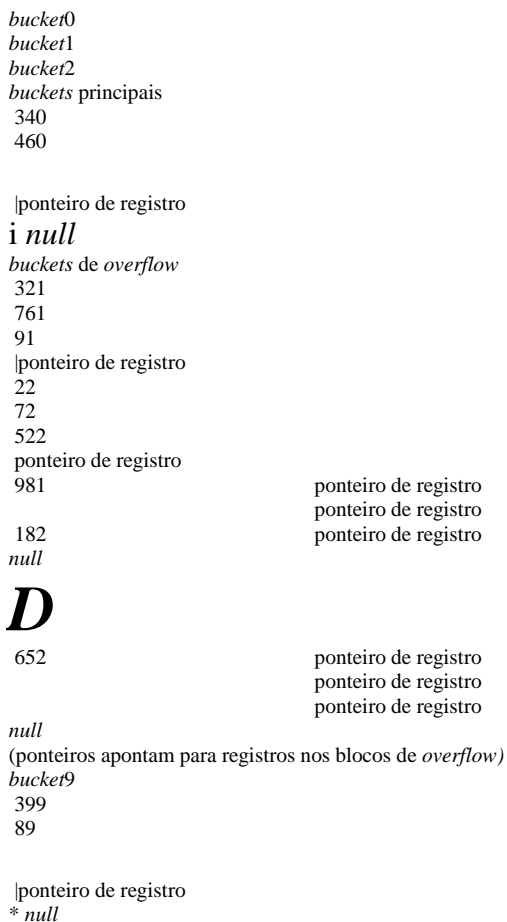


FIGURA 13.10 Tratamento de overflow em *buckets* por encadeamento.

O *hashing* fornece o acesso mais rápido possível para a recuperação de um registro arbitrário, dado o valor de seu campo de *hash*. Embora as melhores funções *hash* não mantenham registros na ordem dos valores do campo de *hash*, algumas funções — chamadas funções **que preservam a ordem** — o fazem. Um exemplo simples de uma função *hash* que preserva uma ordenação é tomar os três dígitos mais à esquerda do campo do número da fatura como endereço *hash* e manter os registros classificados

pelo número da fatura, dentro de cada *bucket*. Um outro exemplo é usar um número inteiro como chave de *hash* diretamente como um índice para um arquivo relativo, se os valores da chave de *hash* preencherem um intervalo em particular; por exemplo, se os códigos dos empregados de uma determinada empresa forem determinados como 1, 2, 3... até o número total de empregados, podemos usar a função *hash* identidade para manter a ordem. Infelizmente, isso funciona apenas se as chaves forem geradas, na aplicação em questão, segundo alguma ordem.

O esquema *hashing* descrito é chamado *hashing* estático porque um número fixo M de *buckets* será alocado. Isso pode ser um sério inconveniente para arquivos dinâmicos. Suponha que tenhamos alocado M *buckets* para o espaço de endereço, que seja m o número máximo de registros que cabem em um *bucket*; então no máximo $(m * M)$ registros caberão no espaço alocado. Se o número de registros se revelar substancialmente menor que $(m * M)$, teremos muito espaço sem uso. Porém, se o número de registros aumentar substancialmente acima de $(m * M)$, haverá numerosas colisões, e a recuperação de registros se tornará mais lenta por conta das longas listas de registros de *overflow*. Em ambos os casos, talvez haja necessidade de troca: o número M de blocos alocados e, então, utilizar uma nova função *hash* (baseada no novo valor de M) para redistribuição dos registros. Essas reorganizações podem consumir bastante tempo para arquivos grandes. Organizações de arquivos dinâmicas mais recentes, baseadas em *hashing*, permitem que o número de *buckets* varie dinamicamente apenas com reorganizações localizadas (Seção 13.8.3).

Quando se utiliza o *hashing* externo, a pesquisa de um registro pelo valor de algum campo diferente do campo de *hash* é tão dispendiosa quanto no caso de um arquivo desordenado. A exclusão de registros pode ser implementada por meio da remoção do registro de seu *bucket*. Se o *bucket* possui um encadeamento de *overflow*, podemos transferir um dos registros de *overflow* para o *bucket* em substituição ao registro excluído. Se o registro a ser excluído já estiver no *overflow*, simplesmente o removemos da lista encadeada. Observe que remover um registro de *overflow* implica nos mantermos informados das posições vazias no *overflow*. Isso é feito facilmente por meio da manutenção de uma lista encadeada de localizações de *overflow* sem uso.

A modificação do valor de um campo de um registro depende de dois fatores: (1) da condição de pesquisa para localizar o registro e (2) do campo a ser modificado. Se a condição de pesquisa for uma comparação de igualdade sobre o campo de *hash*, poderemos localizar o registro eficientemente utilizando a função *hash*; caso contrário, deveremos realizar uma pesquisa linear. Um campo diferente do campo de *hash* pode ser modificado alterando o registro e reescrevendo-o no mesmo *bucket*. A modificação de um campo de *hash* significa que o registro pode ser transferido para um outro *bucket*, o que requer a exclusão do registro antigo seguida da inclusão do registro modificado.

13.8.3 Técnicas Hashing que Permitem a Expansão de Arquivos Dinâmicos

A maior desvantagem do esquema de *hashing* estático, que acabamos de ver, é que o espaço de endereços *hash* é fixo. Desse modo, é difícil expandir ou diminuir o arquivo dinamicamente. Os esquemas descritos nesta seção tentam reparar essa situação. O primeiro esquema — *hashing* extensível — armazena uma estrutura de acesso adicional ao arquivo e, por isso, é de algum modo similar à indexação (Capítulo 6). A principal diferença é que a estrutura de acesso é baseada nos valores resultantes após a aplicação da função *hash* no campo de pesquisa. Na indexação, a estrutura de acesso é baseada nos valores do próprio campo de pesquisa. A segunda técnica, chamada *hashing* linear, não exige estruturas de acesso adicionais.

Esses esquemas de *hashing* tiram vantagem do fato de que o resultado da aplicação de uma função *hash* é um número inteiro não negativo, podendo ser representado como número binário. A estrutura de acesso é construída sobre a representação binária do resultado da função *hash*, que é uma cadeia de bits. Chamamos esse número valor *hash* de um registro. Os registros são distribuídos entre os *buckets*, tendo como base os valores dos primeiros bits de seus valores *hash*.

Hashing Extensível. No *hashing* extensível, um tipo de diretório — um vetor de 2 endereços de *bucket* — é mantido, onde d é chamado profundidade global do diretório. O valor inteiro correspondente aos primeiros (de mais alta ordem) d bits de um valor *hash* é utilizado como um índice de vetor para determinar uma entrada no diretório, e o endereço naquela entrada determina o *bucket* no qual os registros correspondentes serão armazenados. Entretanto, não é necessário que haja um *bucket* diferente para cada uma das 2 localizações. Diversas localizações de diretório com os mesmos primeiros d -bits para seus valores *hash* podem conter o mesmo endereço de *bucket* se todos os registros que a função *hash* levar para essas localizações couberem em um único *bucket*. Uma profundidade local d' — armazenada em cada *bucket* — especifica o número de bits no qual os conteúdos dos *buckets* são baseados. A Figura 13.11 mostra um diretório com profundidade global $d = 3$.

O valor de d pode ser aumentado ou diminuído uma unidade por vez, assim duplicando ou dividindo ao meio o número de entradas no vetor diretório. A duplicação é necessária se ocorrer um *overflow* em um *bucket* cuja profundidade local d' seja igual à profundidade global d . A divisão ao meio ocorre se $d > d'$ para todos os *buckets* após a ocorrência de algumas exclusões. A maioria das recuperações de registro exige dois acessos a blocos: um para o diretório e outro para o *bucket*.

13.8 Técnicas de *Hashing* 315
BUCKETS DO ARQUIVO DE DADOS
profundidade local de cada *bucket*
|d'|*3

/ j/1d' = 3

	DIRETÓRIO	/	^~d'-a
000		^~	
001	S		
010	.-''		
011	•^		
100		i'' 1 1 n	

110 V.

111

profundidade
global d = 3 \ |d* = 3

|d' = 3

bucket para os registros cujos valores *hash* começam com 000
bucket para os registros cujos valores *hash* começam com 001
bucket para os registros cujos valores *hash* começam com 01
bucket para os registros cujos valores *hash* começam com 10
bucket para os registros cujos valores *hash* começam com 110
bucket para os registros cujos valores *hash* começam com 111

FIGURA 13.11 Estrutura do esquema *hashing* extensível.

Para ilustrar a divisão do *bucket*, suponha que um novo registro incluído cause *overflow* no *bucket* cujos valores *hash* começam com 01 — o terceiro *bucket* da Figura 13.11. Os registros serão distribuídos entre dois *buckets*: o primeiro contém todos os registros cujos valores *hash* comecem com 010, e o segundo, aqueles cujos valores *hash* comecem com 011. Agora as duas localizações do diretório para 010 e 011 apontam para dois novos e distintos *buckets*. Antes da divisão, elas apontavam para o mesmo *bucket*. A profundidade local d' dos dois novos *buckets* é 3, que é uma unidade a mais que a profundidade local do *bucket* antigo.

Se um *bucket*, no qual ocorreu um *overflow* e que sofreu divisão, possuir profundidade local d' igual à profundidade global d do diretório, então o tamanho do diretório deve agora ser duplicado de forma que possamos usar um bit a mais para distinguir os dois novos *buckets*. Por exemplo, se no *bucket* dos registros, cujos valores *hash* comecem com 111, da Figura 13.11, ocorrer um *overflow*, os dois novos *buckets* precisarão de um diretório com profundidade global $d = 4$, porque os dois *buckets* agora estão rotulados com 1110 e 1111 e, por isso, suas profundidades locais serão ambas iguais a 4. O tamanho do diretório é, por isso, duplicado, e cada uma das outras localizações originais no diretório também serão divididas em duas localizações, e ambas terão o mesmo valor de ponteiro que a localização original.

A principal vantagem do *hashing* extensível, e o que o torna atraente, é que o desempenho do arquivo não degrada conforme o arquivo cresce, em oposição ao *hashing* externo estático, no qual as colisões aumentam e o encadeamento correspondente causa acessos adicionais. Além disso, nenhum espaço será alocado no *hashing* extensível para crescimento futuro, mas *buckets* adicionais podem ser alocados dinamicamente se necessário. A sobrecarga de espaço para a tabela diretório é insignificante. O tamanho máximo do diretório será 2^k , onde k é o número de bits no valor *hash*. Uma outra vantagem, na maioria dos casos, é que a divisão do *bucket* causará menor número de reorganizações, uma vez que apenas os registros de um *bucket* serão redistribuídos para os dois novos *buckets*. A única reorganização mais dispendiosa será a que for feita quando o diretório tiver de ser duplicado (ou dividido ao meio). Uma desvantagem será que o diretório deverá ser pesquisado antes que os próprios *buckets* sejam acessados, resultando em dois acessos a blocos em vez de um único no caso de *hashing* estático. Essa perda de desempenho é considerada menor — por isso o esquema é preferido em arquivos dinâmicos.

Hashing Linear. A idéia do *hashing linear* é permitir que um arquivo *hash* expanda ou diminua seu número de *buckets* dinamicamente, *sem* necessitar de um diretório. Suponha que o arquivo comece com M *buckets* numerados de $0, 1, \dots, M-1$ e use a função $hash \ mod \ h(K) = K \ mod \ M$; esta função *hash* é chamada função *hash* inicial h_0 . O *overflow* por causa de colisões ainda será necessário e poderá ser tratado por meio da manutenção de cadeias individuais de *overflow* para cada *bucket*. Entretanto, quando uma colisão levar a um registro de *overflow* em *qualquer bucket* do arquivo, o *primeiro bucket* do arquivo — *bucket 0* — será dividido em dois *buckets*: o *bucket 0* original e o novo *bucket M* no final do arquivo. Os registros, originalmente posicionados no *bucket 0*, serão distribuídos entre os dois *buckets* segundo uma função *hash* diferente: $h_{i+1}(K) = K \ mod \ 2M$. Uma propriedade fundamental das duas funções *hash* h e h_{i+1} é que quaisquer registros que forem levados pela função *hash* ao *bucket 0* pela função h , serão levados por h_{i+1} ao *bucket 0* ou ao *bucket M*; isso será necessário para que o *hashing linear* funcione.

Mais adiante, conforme colisões levem registros ao *overflow*, *buckets* adicionais são criados a partir de divisão em ordem linear $1, 2, 3, \dots$. A medida que ocorrem *overflows*, todos os *buckets* $0, 1, \dots, M-1$ originais do arquivo terão sido divididos, assim, o arquivo passa a possuir $2M$ em vez de M *buckets*, e todos os *buckets* usam a função *hash* h_{j+1} . Por isso, os registros no *overflow* serão eventualmente redistribuídos para os *buckets* comuns, usando a função h_{i+j} por meio de uma *divisão tardia* de seus *buckets*. Não há diretório; apenas um valor n — que inicialmente recebe o valor 0 e é incrementado, em uma unidade, quando uma divisão ocorre —, que é necessário para determinar quais *buckets* foram divididos. Para recuperar um registro com valor K na chave de *hash*, primeiro aplica-se a função h_i a K ; se $h_i(K) < n$, então se aplica a função h_{i+1} em K , porque o *bucket* já estará dividido. Inicialmente, $n = 0$, indicando que a função h , se aplica a todos os *buckets*; n cresce linearmente conforme os *buckets* são divididos.

Quando, após ser incrementado, $n = M$, isso significa que todos os *buckets* originais foram divididos e que a função h_{i+1} se aplica a todos os registros do arquivo. Neste ponto, n é reinicializado com 0 (zero), e quaisquer novas colisões que causem *overflow* levarão ao uso de uma nova função *hash* $h_{i+2}(K) = K \ mod \ 4M$. Em geral, a seqüência de funções *hash* $h_{i+j}(K) = K \ mod \ (2^{j+1}M)$ é utilizada, na qual $j = 0, 1, 2, \dots$; uma nova função *hash* h_{j+1} será necessária quando todos os *buckets* $0, 1, \dots, (2^jM) - 1$ tiverem sido divididos e n for reinicializado com 0 . A pesquisa por um registro com valor de chave *hash* K é dada pelo Algoritmo 13.3.

Algoritmo 13.3: Procedimento de pesquisa para o *hashing linear*.

```
if  $n = 0$ 
then  $m \leftarrow h_j(K)$  (*  $m$  é o valor hash de um registro com chave de hash  $K^*$ ) else begin  $m \leftarrow h_j(K)$ ;
if  $m < n$  then  $m \leftarrow h_{j+1}(K)$  end; buscar o bucket cujo valor hash é  $m$  (e seu overflow, se existir);
```

A divisão pode ser controlada pela monitoração do fator de carga do arquivo em vez de realizar a divisão sempre que um *overflow* ocorrer. Em geral, o **fator de carga do arquivo** I pode ser definido como $I = r/(bfr * N)$, onde r é o número corrente de registros do arquivo, bfr é o número máximo de registros que cabem em um *bucket* e N é o número corrente de *buckets* do arquivo. Os *buckets* que foram divididos poderão também ser recombinados se a carga do arquivo cair abaixo de um certo limiar. Os blocos são combinados linearmente e N decrementado apropriadamente. A carga do arquivo pode ser utilizada para disparar divisões e combinações de modo que a carga do arquivo poderá ser mantida dentro de uma faixa desejada. As divisões podem ser disparadas quando a carga exceder um certo limiar — digamos, $0,9$ — e as combinações podem ser iniciadas quando a carga cair abaixo de um outro limiar — digamos, $0,7$.

13.9 OUTRAS ORGANIZAÇÕES PRIMÁRIAS DE ARQUIVO

13.9.1 Arquivos de Registros Mistos

As organizações de arquivos que estudamos até agora supõem que todos os registros de um determinado arquivo sejam do mesmo tipo. Os registros podem ser de EMPREGADOS, PROJETOS, ALUNOS ou DEPARTAMENTOS, mas cada arquivo conterá registros de um único tipo. Na maioria das aplicações de bancos de dados, encontramos situações em que vários tipos de entidades estão inter-relacionadas de várias formas, conforme vimos no Capítulo 3. Os relacionamentos entre registros de vários arquivos

13.10 Acesso Paralelo em Disco Usando a Tecnologia RAID 317

podem ser representados por **campos de conexão**. Por exemplo, um registro de ALUNO pode ter um campo de conexão DEP_ESPECIALIZACAO cujo valor é o nome do DEPARTAMENTO onde o aluno está se especializando. Esse campo DEP_ESPECIALIZACAO se *refere* à entidade DEPARTAMENTO, que deve ser representada por um registro próprio no arquivo DEPARTAMENTO. Se desejarmos recuperar os valores de campo de dois registros relacionados, deveremos recuperar primeiramente um dos registros. Então poderemos usar o valor de seu campo de conexão para recuperar o registro relacionado no outro arquivo. Por isso, os relacionamentos são implementados por **referências lógicas de campos** entre registros de diferentes arquivos.

As organizações de arquivo em SGBDs orientados por objeto, bem como em sistemas legados, tais como SGDBs hierárquicos e SGDBs de rede, freqüentemente implementam relacionamentos entre registros por meio de **relacionamentos físicos** obtidos por meio da adjacência física (ou *clustering*) dos registros relacionados ou por ponteiros físicos. Essas organizações de arquivo normalmente atribuem uma **área** em disco para manter os registros de mais de um tipo, de modo que registros de tipos diferentes possam ficar **fisicamente agrupados (clustered)** no disco. Se for esperada a utilização freqüente de um determinado relacionamento, a implementação física do relacionamento poderá aumentar a eficiência do sistema na recuperação de registros relacionados. Por exemplo, se uma consulta para recuperar o registro de um DEPARTAMENTO, com todos os registros dos ALUNOS que nele se especializam, for muito freqüente, seria desejável posicionar cada registro de DEPARTAMENTO e seu grupo (*cluster*) de registros de ALUNO de maneira contígua, no disco, em um arquivo misto. O conceito de **agrupamento (clustering) físico** de tipos objeto é usado em SGBDs orientados a objeto para armazenar, juntos, objetos relacionados em um arquivo misto. Para distinguir os registros em um arquivo misto, cada registro possui — além dos valores de seus campos — um campo de **tipo de registro**, que especifica o tipo do registro. Normalmente esse é o primeiro campo de cada registro e é utilizado pelo software do sistema para determinar o tipo de registro que ele está prestes a processar. Usando a informação do catálogo, o SGBD pode determinar os campos de cada tipo de registro e seus tamanhos, com o objetivo de interpretar os valores dos dados no registro.

13.9.2 Árvores-B (B-Trees) e Outras Estruturas de Dados como Organizações Primárias

Outras estruturas de dados podem ser usadas como organizações primárias de arquivo. Por exemplo, se tanto o tamanho do registro quanto o número de registros em um arquivo forem pequenos, alguns SGDBs oferecem uma opção de estrutura de dados de árvore-B como organização primária de arquivo. Descreveremos as árvores-B na Seção 14-3.1, quando discutiremos o uso da estrutura de dados de árvore-B para indexação. Em geral, qualquer estrutura de dados que possa ser adaptada às características dos dispositivos de disco poderá ser usada como uma organização de arquivo primária para a disposição de registros no disco.

13.10 ACESSO PARALELO EM DISCO USANDO A TECNOLOGIA RAID

Com o crescimento exponencial da capacidade e do desempenho de dispositivos semicondutores e de memórias, microprocessadores mais rápidos, com memórias cada vez maiores, estão continuamente se tornando mais usuais. Para acompanhar esse crescimento, é natural esperar que a tecnologia de armazenamento secundário também deva caminhar para manter desempenho e confiabilidade compatíveis com a tecnologia de processadores.

Um grande avanço da tecnologia de armazenamento secundário é representado pelo desenvolvimento do RAID, que originalmente significava **Redundant Arrays of Inexpensive Disks (Vetores Redundantes de Discos Baratos)**. Ultimamente o I de RAID é dito *Independent* (Independente). A idéia de RAID recebeu um endosso muito positivo da indústria, e até se desenvolveu um conjunto elaborado de arquiteturas RAID alternativas (de RAID nível 0 até 6). Abaixo, destacamos as principais características dessa tecnologia.

O principal objetivo de RAID é proporcionar melhoria de desempenho de modo a equiparar as taxas, muito diferentes, entre os discos e as memórias de microprocessadores. Enquanto a capacidade das RAMs quadruplica a cada dois ou três anos, os *tempos de acesso* aos discos aumentam menos de 10% ao ano, e as taxas de *transferência* dos discos aumentam aproximadamente 20% ao ano. A *capacidade* dos discos está de fato melhorando em mais de 50%; entretanto, a melhoria da velocidade e do tempo de acesso são de magnitudes muito inferiores. A Tabela 13.3 mostra as tendências na tecnologia de discos, em termos dos valores dos parâmetros em 1993, bem como as taxas de melhoria, por meio dos valores desses parâmetros em 2003.

11 O conceito de chaves estrangeiras do modelo relacional (Capítulo 5) e as referências entre objetos dos modelos orientados por objeto (Capítulo 20) são exemplos de campos de conexão.

12 Essa taxa foi prevista por Gordon Bell em 40% a cada ano entre 1974 e 1984; agora se supõe que exceda 50% ao ano.

318 Capítulo 13 Armazenamento em Disco, Estruturas Básicas de Arquivos e *Hashing***TABELA 13.3** Tendências em tecnologia de discos**Valores dos Parâmetros em 1993*****Taxa Histórica de Melhoria Por Ano (%)*****Valores Atuais (2003)***

Densidade de área Densidade linear Densidade entre trilhas Capacidade (formato 3,5") Taxa de transferência Tempo de pesquisa
 50-150 Mbits/polegada 40.000-60.000 bits/polegada 1.500-3.000 trilhas/polegada 100-2.000 MB 3-4 MB/s 7-20 ms

27 13

10

27 11

36 Gbits/polegada 570 Kbits/polegada 64.000 trilhas/polegada 146 GB 43-78 MB/s 3,5-6 ms

*Fonte: Chen, Lee; Gibson, Katz e Patterson (1994), *ACM Computing Surveys*, Vol. 26, n° 2 (junho, 1994) —

Reprodução autorizada.

*Fonte: Unidades de disco rígido modelos Ultrastar 36XP e 18ZX da IBM.

Há uma segunda disparidade qualitativa originada pelos microprocessadores especiais que servem a novas aplicações para processamento de vídeo, áudio, imagens e dados espaciais (capítulos 24 e 29: detalhes dessas aplicações), com a correspondente falta de acesso rápido a grandes conjuntos de dados compartilhados.

A solução natural é usar um grande vetor de pequenos discos independentes, atuando como um único disco lógico de mais alto desempenho. Um conceito, que usa o *paralelismo* para melhorar o desempenho do disco, chamado **striping** (separação em tiras) **de dados**, é utilizado. O striping de dados distribui os dados de maneira transparente por múltiplos discos para fazê-los parecer como se fossem um único disco grande e rápido. A Figura 13.12 mostra um arquivo distribuído ou separado (*striped*) em quatro discos. O striping de dados melhora o desempenho total de entrada e saída, permitindo que diversas entradas e saídas sejam realizadas em paralelo, fornecendo, assim, altas taxas de transferência globais. O *striping* de dados também auxilia o balanceamento de carga entre discos. Além disso, por meio do armazenamento de informações redundantes em discos, usando paridade ou outro código para correção de erros, a confiabilidade pode ser melhorada. Nas seções 13.10.1 e 13.10.2 veremos como o RAID consegue atingir dois importantes objetivos para melhoria de confiabilidade e desempenho. A Seção 13.10.3 analisará as organizações de RAID.

ARQUIVO A

£

disco 0

disco 1

disco 2

disco 3

Ai

^|

H

A₃

A4

FIGURA 13.12 *Striping* de dados. O arquivo A é separado {*striped*} em quatro discos.

13.10.1 Melhoria de Confiabilidade com RAID

Para um vetor de n discos, a probabilidade de falha é n vezes a de um disco. Por isso, caso seja suposto que o MTTF (Mean Time To Failure — Tempo Médio entre Falhas) de uma unidade de disco seja de 200.000 horas, ou cerca de 22,8 anos (valores normais variam até 1 milhão de horas), o MTTF de 100 unidades de disco cai para apenas 2.000 horas, ou 83,3 dias. Manter uma única cópia dos dados em tal vetor de discos causará uma significativa perda de confiabilidade. Uma solução óbvia é empregar a redundância de dados de modo que as falhas de discos sejam toleráveis. As desvantagens são muitas: operações de entrada e saída adicionais para a gravação, processamento adicional para a manutenção da redundância e para a execução de recuperação de erros, além da capacidade de disco adicional para armazenar informações redundantes.

Uma técnica para a introdução da redundância é chamada **espelhamento** (*mirroring* ou *shadowing*). Os dados são escritos de maneira redundante em dois discos físicos idênticos, tratados como um único disco lógico. Quando os dados forem lidos, eles serão recuperados do disco com menores atrasos por fila, tempo de busca e rotacional. Se um disco falhar, o outro será usado até que o primeiro seja reparado. Suponha que o tempo médio para a reparação seja de 24 horas, então o tempo médio para a perda de dados de um sistema de disco espelhado usando 100 discos, com MTTF de 200.000 horas cada, é *de*

13.10 Acesso Paralelo em Disco Usando a Tecnologia RAID 319

$(200.000)/(2 \times 24) = 8,33 \times 10$ horas, que são 95.028 anos. O espelhamento de disco também dobra a taxa de tratamento das solicitações de leitura, uma vez que a leitura poderá ocorrer em ambos os discos. A taxa de transferência de cada leitura, entretanto, continua igual à de um único disco.

Uma outra solução para o problema da confiabilidade é armazenar informações adicionais, que não são normalmente necessárias, mas que podem ser utilizadas para reconstruir informações perdidas em caso de falha de disco. A incorporação da redundância deverá levar em consideração dois problemas: (1) a seleção de uma técnica para a obtenção das informações redundantes e (2) a seleção de um método para a distribuição das informações redundantes pelo vetor de discos. O primeiro problema é tratado utilizando-se códigos de correção de erros envolvendo bits de paridade, ou códigos especializados, como os códigos Hamming. Sob o esquema da paridade, considera-se que um disco redundante possua a soma de todos os dados dos outros discos. Quando um disco falhar, a informação perdida pode ser construída por um processo similar ao da subtração.

Para o segundo problema, as duas principais abordagens são armazenar a informação redundante em um pequeno número de discos ou distribuí-la uniformemente por todos os discos. A última opção resulta em melhor balanceamento de carga. Os diferentes níveis de RAID optam por uma combinação dessas opções para implementar a redundância e, assim, melhorar a confiabilidade.

13.10.2 Melhoria de Desempenho com RAID

O vetor de discos emprega a técnica de *striping* de dados para obter taxas de transferência mais altas. Observe que os dados podem ser lidos ou escritos em apenas um bloco por vez, assim uma transferência típica contém 512 bytes. O striping de discos pode ser aplicado a uma granularidade mais fina, por meio da quebra de um byte de dados em bits e espalhando os bits em diferentes discos. Assim, o **striping de dados no nível de bits** consiste na divisão de um byte de dados e da gravação do bit j no j -ésimo disco. Com

bytes de 8 bits, oito discos físicos podem ser considerados como um único disco lógico com um aumento de oito vezes na taxa de transferência de dados. Cada disco participa de cada pedido de entrada e saída, e a quantidade total de dados lidos por pedido é oito vezes maior. O *striping* de dados no nível de bits pode ser generalizado para um número de discos que seja um múltiplo ou um divisor de oito. Assim, em um vetor de quatro discos, o bit n vai para o disco $(n \bmod 4)$.

A granularidade dos dados intercalados pode ser maior que um bit; por exemplo, blocos de um arquivo podem ser separados (*stripped*) pelos discos, surgindo o ***striping no nível de blocos***. A Figura 13.12 mostra o *striping* de dados no nível de blocos, supondo-se que o arquivo de dados contivesse quatro blocos. Com o *striping* no nível de blocos, diversos pedidos independentes que acessem blocos únicos (pequenos pedidos) poderão ser atendidos em paralelo por discos separados, diminuindo, assim, o tempo de fila de pedidos de entrada e saída. Os pedidos que acessem diversos blocos (grandes pedidos) podem ser realizados em paralelo, reduzindo assim seus tempos de resposta. Geralmente, quanto maior o número de discos em um vetor, maior o potencial de benefício no desempenho. Entretanto, supondo falhas independentes, o vetor de discos de 100 discos coletivamente tem 1/100 da confiabilidade de um único disco. Assim, a redundância por meio de códigos de correção de erros e espelhamento de discos é necessária para prover confiabilidade e alto desempenho.

13.10.3 Organizações e Níveis de RAID

Foram definidas diferentes organizações de RAID com base em diferentes combinações dos dois fatores de granularidade de dados intercalados (*striping*) e padrão utilizados para calcular a informação redundante. Na proposta inicial, foram propostos níveis de 1 a 5 de RAID, e dois outros níveis — 0 e 6 — foram adicionados posteriormente.

O RAID nível 0 usa *striping* de dados, não possui redundância e, assim, apresenta o melhor desempenho de gravação, uma vez que as atualizações não precisam ser duplicadas. Entretanto, seu desempenho de leitura não é tão bom quanto no RAID nível 1, que usa espelhamento de discos. Neste, a melhoria de desempenho é possível por meio do agendamento de um pedido de leitura no disco com menor expectativa de atrasos de busca e rotacional. O RAID nível 2 usa a redundância *memory-style* (estilo memória) por meio do uso de códigos Hamming, os quais contêm bits de paridade para distintos subconjuntos de componentes sobrepostos. Assim, em uma dada versão desse nível, três discos redundantes bastam para aplicar a redundância aos quatro discos originais, enquanto com espelhamento — como no nível 1 — seriam necessários quatro discos. O nível 2 inclui tanto a detecção quanto a correção de erros, embora a detecção geralmente não seja necessária porque discos quebrados são detectados por si próprios.

O RAID nível 3 usa um único disco de paridade, contando com a controladora de discos para identificar qual deles falhou. Os níveis 4 e 5 usam *striping* de dados no nível de blocos, com o nível 5 distribuindo dados e informação de paridade por todos os discos. Finalmente, o RAID nível 6 aplica o assim chamado esquema de redundância $P + Q$ usando os códigos

13 As fórmulas para o cálculo de MTTF aparecem em Chen *et al.* (1994).

320 Capítulo 13 Armazenamento em Disco, Estruturas Básicas de Arquivos e *Hashing*

Reed-Solomon para proteção contra até duas falhas de discos por meio do uso de apenas dois discos redundantes. Os sete níveis de RAID (0 até 6) são ilustrados esquematicamente na Figura 13.13.

Não-redundante (RAID Nível 0)

Espelhamento (RAID Nível 1)

ECC (código de correção de erro) *memory-style* (RAID Nível 2)

Paridade com Bits Intercalados (RAID Nível 3)

Paridade com Blocos Intercalados (RAID Nível 4)

Paridade Distribuída com Blocos Intercalados (RAID Nível 5)

Redundância P+Q (RAID Nível 6)

FIGURA 13.13 Diversos níveis de RAID. Fonte: Chen, Lee; Cibson, Katze Patterson (1994), *ACM Computing Surveys*, Vol. 2 n° 2 (junho, 1994). Reprodução autorizada.

A reconstrução, em caso de falha de disco, é mais fácil no RAID nível 0. Outros níveis exigem a reconstrução de um disco que falhou por meio da leitura de diversos discos. O nível 1 é usado em aplicações críticas, como o armazenamento de I/O de transações. Os níveis 3 e 5 são os preferidos para o armazenamento de grandes volumes, com o nível 3 fornecendo taxas de transferência mais altas. Atualmente, a utilização mais comum da tecnologia RAID usa o nível 0 (com *striping*), o nível 1 (espelhamento) e o nível 5 com uma unidade de disco adicional para paridade. Projetistas de configuração de um RAID para um determinado conjunto de aplicações precisam enfrentar muitas decisões de projeto, tais como o nível de RAID, o número de discos, a escolha entre os esquemas de paridade e o agrupamento para o *striping* no nível de blocos. Foram realizados estudos detalhados sobre o desempenho em pequenas leituras e escritas (referentes aos pedidos de entrada e saída para uma única unidade *striping*) e em grandes leituras e escritas (referentes aos pedidos de entrada e saída para uma única unidade *striping* em cada disco, em um grupo de correção de erros).

13.11 Área de Armazenamento em Rede 321

13.11 ÁREA DE ARMAZENAMENTO EM REDE

Com o rápido crescimento do comércio eletrônico, dos sistemas *Enterprise Resource Planning* (ERP — Sistemas de Planejamento de Recursos Empresariais), que integram as aplicações por todas as áreas das organizações, e dos *data warehouses*, que mantêm informações históricas agregadas (Capítulo 27), a demanda por armazenamento cresceu consideravelmente. Para as atuais organizações direcionadas para a Internet, tornou-se necessário mudar de uma operação com centro de dados (*data center*) fixo e estático para uma infra-estrutura mais flexível e dinâmica, dado seus requisitos de processamento de informação. O custo total para manutenção de todos os dados está crescendo tão rapidamente que, em algumas situações, o custo de gerenciamento do servidor conectado (*attached*) ao armazenamento é mais caro que o custo do próprio servidor. Além disso, o custo da compra do armazenamento é apenas uma pequena fração — apenas 10% a 15% do custo total do gerenciamento do armazenamento. Muitos usuários de sistemas RAID não podem usar efetivamente sua capacidade porque precisam estar conectados (*attached*) de uma maneira fixa a um ou mais servidores. Por essa razão, grandes organizações estão adotando um conceito chamado Área de Armazenamento em Redes (SAN — *Storage Area Networks*). Em uma SAN, periféricos de armazenamento *on-line* são configurados como nós em uma rede de alta velocidade e podem ser conectados (*attached*) e desconectados (*detached*) dos servidores de maneira bastante flexível. Diversas empresas têm surgido como provedoras de SAN e fornecem suas próprias topologias. Elas permitem que os sistemas de armazenamento sejam posicionados a grandes distâncias dos servidores e proporcionam diferentes opções de desempenho e conectividade. As aplicações de gerenciamento de armazenamento existentes atualmente podem ser portadas para a configuração SAN utilizando redes de Canais de Fibras Ópticas, que encapsulam o protocolo SCSI legado. Como resultado, dispositivos conectados à SAN aparecem como se fossem dispositivos SCSI.

Algumas das alternativas de arquitetura atuais para a SAN incluem as seguintes: conexões 'ponto-a-ponto' entre servidores e sistemas de armazenamento por meio de canal de fibra óptica; uso de um *switch* com canal de fibra óptica para conectar múltiplos sistemas RAID, bibliotecas de fitas etc. a servidores; uso de *hubs* e *switches* com canal de fibra óptica para conectar servidores e sistemas de armazenamento em diferentes configurações.

As organizações podem mudar lentamente, das tecnologias mais simples para as mais complexas, por meio do acréscimo de servidores e dispositivos de armazenamento conforme necessário. Não forneceremos mais detalhes aqui porque eles diferem entre os fornecedores de SANs. As principais vantagens alegadas são as seguintes:

- Conectividade 'muitos-para-muitos' flexível entre servidores e dispositivos de armazenamento usando *hubs* e *switches* com canal de fibra óptica.
- Separação de até dez quilômetros entre um servidor e um sistema de armazenamento usando cabos adequados de fibras ópticas.
- Melhor capacidade de isolamento permitindo a adição, sem interrupção, de novos periféricos e servidores.

As SANs têm crescido muito rapidamente, mas ainda enfrentam muitos problemas, tais como a combinação de opções de armazenamento de diferentes fornecedores e negociações envolvendo os padrões de software e hardware de gerenciamento de armazenamento. A maior parte das grandes empresas está avaliando a SAN como uma opção viável para o armazenamento de bancos de dados.

13.12 RESUMO

Começamos este capítulo analisando as características das hierarquias de memórias e depois nos concentramos nos dispositivos de armazenamento secundário. Em particular, enfocamos os discos magnéticos porque eles são mais freqüentemente usados para o armazenamento *on-line* de arquivos de bancos de dados.

Os dados em discos são armazenados em blocos; o acesso a um bloco de disco é dispendioso por causa do tempo de busca (pesquisa), do atraso rotacional e do tempo de transferência do bloco. O *buffering* duplo pode ser usado quando blocos de disco consecutivos forem acessados, a fim de reduzir a média do tempo de acesso ao bloco. Outros parâmetros de disco são discutidos no Apêndice B. Apresentamos diversas maneiras de armazenar os registros de um arquivo em disco. Os registros de um arquivo são agrupados em blocos de disco e podem ser de tamanho fixo ou variável, *spanned* e *não-spanned*, e do mesmo tipo de registro ou de tipos mistos. Vimos o cabeçalho do arquivo, o qual descreve os formatos do registro e mantém a informação dos endereços no disco dos blocos do arquivo. A informação no cabeçalho do arquivo é utilizada pelo software do sistema que acessa os registros do arquivo.

322 Capítulo 13 Armazenamento em Disco, Estruturas Básicas de Arquivos e *Hashing*

Apresentamos um conjunto de comandos típicos para o acesso individual dos registros de arquivos e discutimos o conceito de registro corrente de um arquivo. Analisamos como complexas condições de busca de registro são transformadas em condições de pesquisa mais simples, usadas para localizar os registros no arquivo.

Três organizações primárias de arquivo foram, então, vistas: desordenado, ordenado e *hashed*. Arquivos desordenados exigem uma pesquisa linear para localizar registros, mas a inclusão de registros é muito simples. Discutimos o problema da exclusão e o uso de marcadores de exclusão.

Os arquivos ordenados diminuem o tempo necessário para a leitura dos registros na ordem do campo de classificação. O tempo necessário para pesquisar um registro arbitrário, dado o valor de seu campo-chave de classificação, também é reduzido se a pesquisa binária for utilizada. Entretanto, a manutenção dos registros na ordem torna a inclusão muito dispendiosa; por isso, foi discutida a técnica do uso de um arquivo de *overflow* para reduzir o custo da inclusão de registros. Os registros de *overflow* são periodicamente incorporados ao arquivo mestre durante a reorganização do arquivo.

O *hashing* proporciona acesso muito rápido a um registro arbitrário de um arquivo, conhecido o valor de sua chave de hash. O método mais conveniente para o *hashing* externo é a técnica de *bucket*, na qual cada *bucket* corresponde a um ou mais blocos adjacentes. As colisões que causam *overflow* no *bucket* são tratadas por encadeamento. O acesso a qualquer campo que não seja o campo de hash é tão lento quanto o acesso ordenado. Depois vimos duas técnicas de *hash* para arquivos que expandem e reduzem o número de registros dinamicamente — nominalmente, *hashing* extensível e *hashing* linear.

Discutimos brevemente outras possibilidades para organizações primárias de arquivos, tais como árvores-B (B-trees) e arquivos de registros mistos, que implementam fisicamente relacionamentos entre registros de diferentes tipos como parte da estrutura de armazenamento. Finalmente, revimos os avanços recentes na tecnologia de discos representada pelo RAID (*Redundant Arrays of Inexpensive [or Independent] Disks* — Vetores Redundantes de Discos Baratos [ou Independentes]).

Questões para Revisão

- 13.1 Qual a diferença entre armazenamento primário e armazenamento secundário?
- 13.2 Por que são usados discos e não fitas para armazenar arquivos de bancos de dados *on-line*?
- 13.3 Defina os seguintes termos: *disco*, *conjunto de discos*, *trilha*, *bloco*, *cilindro*, *setor*, intervalo entre blocos (*interblock gap*), *cabeçote de leitura/escrita*.
- 13.4 Discuta o processo de inicialização de um disco.
- 13.5 Discuta o mecanismo usado para ler e para escrever os dados no disco.
- 13.6 Quais os componentes de um endereço de bloco de disco?
- 13.7 Por que o acesso a um bloco de disco é dispendioso? Discuta os componentes de tempo envolvidos no acesso a um bloco de disco.
- 13.8 Descreva o descompasso entre a tecnologia dos processadores e dos discos.
- 13.9 Quais são os principais objetivos da tecnologia RAID? Como ela os alcança?
- 13.10 Como o espelhamento de disco ajuda a melhorar o desempenho? Dê um exemplo quantitativo.
- 13.11 Quais são as técnicas utilizadas para melhorar o desempenho dos discos pelo RAID?
- 13.12 O que caracteriza os níveis da organização RAID?
- 13.13 Como o *buffering* duplo melhora o tempo de acesso ao bloco?
- 13.14 Quais são as razões para ter registros de tamanho variável? Que tipos de caracteres separadores são necessários para cada uma delas?
- 13.15 Discuta as técnicas para a alocação de blocos de arquivo no disco.
- 13.16 Qual a diferença entre uma organização de arquivo e um método de acesso?
- 13.17 Qual a diferença entre arquivos estáticos e dinâmicos?
- 13.18 Quais são as operações típicas *record-at-a-time* (um registro por vez) de acesso a um arquivo? Quais dependem do registro corrente do arquivo?
- 13.19 Discuta as técnicas para exclusão de registros.
- 13.20 Discuta as vantagens e as desvantagens do uso de (a) um arquivo desordenado, (b) um arquivo ordenado e (c) um arquivo *hash* estático com *buckets* e encadeamento. Quais operações podem ser realizadas eficientemente em cada uma dessas organizações e quais operações são dispendiosas?
- 13.21 Discuta as técnicas para possibilitar a expansão ou retração dinâmica de um arquivo *hash*. Quais as vantagens e as desvantagens de cada uma delas?
- 13.22 Para que são usados os arquivos mistos? Quais são os outros tipos de organizações primárias de arquivos?

Exercícios

13.23 Considere um disco com as seguintes características (estes não são parâmetros de nenhuma unidade de disco em particular): tamanho de bloco $B = 512$ bytes; tamanho de intervalo entre blocos $G = 128$ bytes; número de blocos por trilha = 20; número de trilhas por superfície = 400. Um conjunto de discos (*disk pack*) consiste de 15 discos dupla face.

- Qual é a capacidade total de uma trilha, e qual é sua capacidade útil (excluindo os intervalos entre blocos)?
- Qual a quantidade de cilindros?
- Qual é a capacidade total e a capacidade útil de um cilindro?
- Qual é a capacidade total e a capacidade útil de um conjunto de discos?
- Suponha que a unidade de disco rotaciona o conjunto de discos a uma velocidade de 2.400 rpm (revoluções por minuto). Quais são a taxa de transferência (tr) em bytes/ms e o tempo de transferência de bloco (btt) em ms? Qual o atraso rotacional (rd) em ms? Qual a taxa de transferência de volume? (Apêndice B, no site deste livro).
- Suponha que o tempo médio de pesquisa é de 30 ms. Quanto tempo leva (em média) em ms para localizar e transferir um único bloco, dado o endereço do bloco?
- Calcule o tempo médio que levaria para transferir 20 blocos aleatórios, e compare com o tempo que levaria para transferir 20 blocos consecutivos, utilizando-se o *buffering* duplo para economizar tempo de pesquisa e reduzir atraso rotacional.

13.24 Um arquivo possui $r = 20.000$ registros de ALUNO com *tamanho fixo*. Cada registro possui os seguintes campos: NOME (30 bytes), SSN (9 bytes), ENDEREÇO (40 bytes), TELEFONE (9 bytes), DATANASC (8 bytes), SEXO (1 byte), DEP_REFERENCIA (4 bytes), DEP_ESPECIALIZACAO (4 bytes), COD_SALA (4 bytes, número inteiro) e PROGRAMA_DE_GRADUACAO (3 bytes). Um byte adicional é utilizado como marcador de exclusão. O arquivo está armazenado em um disco cujos parâmetros são aqueles dados no Exercício 13.23.

- Calcule o tamanho R do registro em bytes.
- Calcule o fator de divisão de bloco bfr e o número b de blocos do arquivo, supondo uma organização não-spanned.
- Calcule o tempo médio para localizar um registro por meio de uma pesquisa linear no arquivo se (i) os blocos do arquivo estiverem armazenados de maneira adjacente e for utilizado *buffering* duplo; (ii) os blocos do arquivo não estiverem armazenados de maneira adjacente.
- Suponha que o arquivo esteja ordenado por SSN; calcule o tempo para buscar um registro, dado seu valor de SSN por meio da execução de uma pesquisa binária.

13.25 Suponha que apenas 80% dos registros de ALUNO do Exercício 13.24 possuam um valor para TELEFONE, 85% para DEP_REFERENCIA, 15 % para DEP_ESPECIALIZACAO e 90 % para PR0GRAMA_DE_GRADUACAO; e suponha que usemos um arquivo com registros de tamanho variável. Cada registro possui um código de *tipo de campo* de 1 byte para cada campo do registro, mais um marcador de exclusão de 1 byte e um marcador de fim de registro de 1 byte. Suponha que usemos a organização de registros *spanned*, na qual cada bloco possui um ponteiro de 5 bytes para o próximo bloco (esse espaço não é utilizado para armazenamento do registro).

- Calcule o tamanho médio R dos registros em bytes.
- Calcule o número de blocos necessários para o arquivo.

13.26 Suponha que uma unidade de disco possua os seguintes parâmetros: tempo de pesquisa $s = 20$ ms; atraso rotacional $rd = 10$ ms; tempo de transferência de bloco $btt = 1$ mseg; tamanho de bloco $B = 2.400$ bytes; tamanho do intervalo entre blocos $G = 600$ bytes. Um arquivo EMPREGADO possui os seguintes campos: SSN, 9 bytes; ULTIMO_NOME, 20 bytes; PRIMEIRO_NOME, 20 bytes; INICIAL_DO_MEIO, 1 byte; DATANASC, 10 bytes; ENDEREÇO, 35 bytes; TELEFONE, 12 bytes; SSN_SUPERVISOR, 9 bytes; DEPARTAMENTO, 4 bytes; CODIGOCARGO, 4 bytes; *marcador de exclusão*, 1 byte. O arquivo EMPREGADO possui $r = 30.000$ registros de formato de tamanho fixo e blocos não-spanned. Escreva fórmulas apropriadas e calcule os seguintes valores para o arquivo EMPREGADO acima:

- O tamanho de registro R (incluindo o marcador de exclusão). O fator de divisão de blocos bfr e o número de blocos de disco b .
- Calcule o espaço desperdiçado em cada bloco por conta da organização não-spanned.
- Calcule a taxa de transferência tr e a taxa de transferência de volume $\mathcal{L} > tr$ para essa unidade de disco (Anexo B: definições de tr e btr).
- Calcule o *número médio de acessos a blocos* necessários para pesquisar um registro arbitrário no arquivo usando a pesquisa linear.
- Calcule, em ms, o tempo médio necessário para pesquisar um registro arbitrário no arquivo, usando pesquisa linear, se os blocos do arquivo forem armazenados em blocos consecutivos de disco e o *buffering* duplo for utilizado.

Capítulo 13 Armazenamento em Disco, Estruturas Básicas de Arquivos e *Hashing*

f. Calcule, em ms, o tempo médio necessário para pesquisar um registro arbitrário no arquivo, usando pesquisa linear, se os blocos do arquivo não estiverem armazenados em blocos consecutivos de disco.

g. Suponha que os registros estejam classificados segundo algum campo-chave. Calcule o número médio de acessos a blocos e o tempo médio necessários para pesquisar um registro arbitrário no arquivo, usando a pesquisa binária.

13.27 O arquivo PECAS com NUMERO_Peca como chave de hash possui registros com os seguintes valores de NUMERO_Peca: 2369 3760,4692,4871,5659,1821,1074, 7115,1620, 2428,3943,4750,6975,4981,9208. O arquivo usa oito buckets, número de 0 a 7. Cada bucket é um bloco de disco e mantém dois registros. Carregue os registros no arquivo na ordem dada, utilizando a função $hash(K) = K \bmod 8$. Calcule o número médio de acessos a bloco para uma recuperação aleatória baseada em um NUMERO_Peca.

13.28 Carregue os registros do Exercício 13.27 em arquivos hash extensíveis baseados em hashing extensível. Mostre a estrutura de diretório em cada passo e as profundidades local e global. Use a função $hash(K) = K \bmod 128$.

13.29 Carregue os registros do Exercício 13.27 em um arquivo hash extensível usando hashing linear. Comece com um único bloco de disco usando a função $hash(K) = K \bmod 2$, e mostre como o arquivo cresce e como as funções hash mudam conforme os registros são incluídos. Suponha que os blocos sejam sempre divididos quando ocorrer um overflow < mostre o valor de n em cada passo.

13.30 Compare os comandos de arquivo listados na Seção 13.6 com aqueles do método de acesso de arquivo que você está familiarizado.

13.31 Suponha que tenhamos um arquivo desordenado de registros de tamanho fixo que use uma organização de arquivo não-spanned. Esboce algoritmos para a inclusão, a exclusão e a modificação de um registro do arquivo. Declare quais suposições que você fizer.

13.32 Suponha que tenhamos um arquivo ordenado de registros de tamanho fixo e um arquivo de overflow para o tratamento de inclusões. Ambos os arquivos usam registros não-spanned. Esboce algoritmos para a inclusão, a exclusão e a modificação de um registro do arquivo e para a reorganização do arquivo. Declare quaisquer suposições que você faça.

13.33 Você consegue imaginar outras técnicas mais eficientes diferentes do arquivo desordenado de overflow que possam ser usadas para realizar inclusões em um arquivo ordenado?

13.34 Suponha que tenhamos um arquivo hash de registros de tamanho fixo e suponha que o overflow é tratado por encadeamento. Esboce algoritmos para a inclusão, a exclusão e a modificação de um registro do arquivo. Declare quaisquer suposições que você fizer.

13.35 Você consegue imaginar outras técnicas diferentes de encadeamento para tratar o overflow em buckets no hashing extensivo?

13.36 Escreva um pseudocódigo para os algoritmos de inclusão em hashing linear e em hashing extensivo.

13.37 Escreva códigos de programa para acesso a campos individuais dos registros sob cada uma das seguintes circunstâncias. Para cada caso, declare as suposições que você fizer relacionadas a ponteiros, caracteres separadores, e assim por diante. Determine o tipo de informação necessária no cabeçalho do arquivo para que seu código possa ser usado de maneira generalizada em cada caso.

- Registros de tamanho fixo com blocos não-spanned.
- Registros de tamanho fixo com blocos spanned.
- Registros de tamanho variável com campos de tamanho variável e blocos spanned.
- Registros de tamanho variável com grupos de repetição e blocos spanned.
- Registros de tamanho variável com campos opcionais e blocos spanned.
- Registros de tamanho variável que permitam todos os casos dos itens c, d e e.

13.38 Suponha que um arquivo inicialmente contenha $r = 120.000$ registros de $R = 200$ bytes cada em arquivo desordenado (heap). O tamanho do bloco $B = 2.400$ bytes, o tempo médio de pesquisa $s = 16$ ms, a latência rotacional média $rd = 8,3$ ms e o tempo de transferência de bloco $btt = 0,8$ ms. Suponha que um registro seja excluído para cada dois registros incluídos até que o total de registros ativos seja de 240.000.

- Quantas transferências de bloco são necessárias para reorganizar o arquivo?
- Quanto tempo leva para encontrar um registro imediatamente antes da reorganização?
- Quanto tempo leva para encontrar um registro imediatamente após a reorganização?

13.39 Suponha que tenhamos um arquivo sequencial (ordenado) de cem mil registros, no qual cada registro é de 240 bytes. Suponha que $B = 2.400$ bytes, $s = 16$ ms, $rd = 8,3$ ms e $btt = 0,8$ ms. Suponha que queiramos realizar X leituras independentes e aleatórias de registros do arquivo. Poderíamos realizar X leituras aleatórias de blocos ou poderíamos executar uma leitura exaustiva do arquivo inteiro procurando por aqueles X registros. A questão é decidir quando seria mais eficiente realizar uma leitura exaustiva do arquivo inteiro em vez de realizar X leituras individuais aleatórias. Ou

13.12 Resumo | 325

seja, qual o valor de X para que a leitura exaustiva do arquivo seja mais eficiente que as X leituras aleatórias? Desenvolva sua solução como uma função de X .

13.40 Suponha que um arquivo *hash* estático possua inicialmente 600 *buckets* na área primária e que os registros que serão incluídos criem uma área de *overflow* de 600 *buckets*. Se reorganizarmos o arquivo *hash*, poderemos supor que o *overflow* seja eliminado. Se o custo da reorganização do arquivo é o custo das transferências de *bucket* (leitura e escrita de todos os *buckets*), e a única operação de arquivo periódica é a operação de pesquisa, então quantas vezes deveríamos realizar uma pesquisa (com sucesso) para realizar a reorganização de forma eficiente no custo? Ou seja, para que o custo da reorganização e o custo da pesquisa subsequente sejam menores que o custo de pesquisa antes da reorganização. Argumente sua resposta. Suponha $s = 16$ ms, $r = 8,3$ ms, $btt = 1$ ms.

13.41 Suponha que queiramos criar um arquivo *hash* linear com um fator de carga de arquivo de 0,7 e fator de divisão de bloco de 20 registros por *bucket*, o qual deve conter inicialmente 112.000 registros.

- Quantos *buckets* deveríamos alocar na área primária?
- Qual deve ser o número de bits usados para os endereços de *bucket*?

Bibliografia Seleccionada

Em Wiederhold (1983) há uma discussão e uma análise detalhadas dos dispositivos de armazenamento secundário e das organizações de arquivo. Discos ópticos são descritos em Berg e Roth (1989) e analisados em Ford e Christodoulakis (1991). Memória *flash* é discutida por Dippert e Levy (1993). Ruemmler e Wilkes (1994) apresentam um levantamento sobre a tecnologia de disco magnético. A maioria dos livros-texto sobre bancos de dados inclui discussões sobre os materiais apresentados aqui. A maioria dos livros de referência sobre estruturas de dados, inclusive Knuth (1973), discute o *hashing* estático com mais detalhes; Knuth tem uma discussão completa sobre funções *hash* e técnicas de resolução de colisão, bem como da comparação de seus desempenhos. Knuth também traz uma discussão detalhada das técnicas para a classificação de arquivos externos. Livros de referência sobre estruturas de arquivo compreendem Claybrook (1983), Smith e Barnes (1987) e Salzberg (1988); eles discutem outras organizações de arquivos, incluindo arquivos estruturados em árvores (*trees*), e trazem algoritmos detalhados para as operações em arquivos. Outros livros sobre organizações de arquivo incluem Miller (1987) e Livadas (1989). Salzberg *et al.* (1990) descrevem um algoritmo distribuído de classificação externa. Organizações de arquivo com alto grau de tolerância a falhas são descritas por Bitton e Gray (1988) e por Gray *et al.* (1990). Ostripping de disco é proposto por Salem e Garcia Molina (1986). O primeiro artigo sobre *Redundant Arrays of Inexpensive Disks* (RAID — Vetores Redundantes de Discos Baratos) é de Patterson *et al.* (1988). Chen e Patterson (1990) e o excelente levantamento sobre RAID de Chen *et al.* (1994) são referências adicionais. Grochowski e Hoyt (1996) discutem tendências futuras em unidades de disco. Várias fórmulas para a arquitetura de RAID aparecem em Chen *et al.* (1994). Morris (1968) é um artigo antigo sobre *hashing*. *Hashing* extensível é descrito em Fagin *et al.* (1979). *Hashing* linear é descrito por Litwin (1980). *Hashing* dinâmico, que nós não discutimos em detalhes, foi proposto por Larson (1978). Há muitas variações propostas para *hashing* extensível e linear; veja, por exemplo, Cesarini e Soda (1991), Du e Tong (1991), e Ha-cheme Berra (1992). Detalhes sobre dispositivos de armazenamento em disco podem ser encontrados em *sites* de fabricantes; por exemplo, www.seagate.com, www.ibm.com, www.storagetek.com. A IBM possui um centro de pesquisa em tecnologia de armazenamento em IBM Almaden (www.almaden.ibm.com/sst/).

Neste capítulo assumimos que um arquivo já existe e possui alguma organização primária, como a desordenada, a ordenada ou *hash*, que foram descritas no Capítulo 13. Descreveremos **estruturas de acesso** adicionais auxiliares chamadas **índices**, que são usadas para aumentar a velocidade da recuperação de registros na resposta a certas condições de busca. Geralmente, as estruturas de índice fornecem caminhos de acesso secundário, que provêem caminhos de acesso alternativos aos registros sem afetar a disposição física dos registros no arquivo. Elas possibilitam um acesso eficiente aos registros a partir de **campos de indexação** que são usados para construir o índice. Basicamente, *qualquer campo* do arquivo pode ser usado para criar um índice, e podem ser construídos *índices múltiplos* a partir de diferentes campos no mesmo arquivo. É possível uma variedade de índices, em que cada um deles use uma estrutura de dados específica para aumentar a velocidade da busca. Para encontrar um registro ou registros em um arquivo com base em certo critério de seleção, a partir de um campo de indexação, é necessário inicialmente acessar o índice, que aponta para um ou mais blocos do arquivo em que os registros requeridos estão localizados. Os tipos de índice mais predominantes são baseados em arquivos ordenados (índices em nível único) e estruturas de dados de árvores (índices multiníveis, árvores-B). Eles também podem ser construídos em *hashing* ou em outras estruturas de busca de dados.

Descreveremos diferentes tipos de índice ordenados em nível único — primário, secundário e *clustering (agrupamento)* — na Seção 14-1. Tratando um índice de nível único como um arquivo ordenado, é possível desenvolver índices adicionais, onde surge o conceito de índices multiníveis. O ISAM (*Indexed Sequential Access Method* — Método de Acesso Sequencial Indexado) é um esquema popular de indexação baseado nessa idéia. Discutiremos os índices multiníveis na Seção 14.2. Na Seção 14.3 descreveremos as árvores-B e as árvores-B⁺, que são estruturas de dados usadas comumente em SGBDs para implementar dinamicamente índices multiníveis. As árvores-B tornaram-se uma estrutura padrão comumente aceita para a geração de índices sob demanda em SGBDs relacionais. A Seção 14.4 é dedicada aos caminhos alternativos de acesso aos dados baseando-se na combinação de chaves múltiplas. Na Seção 14.5 discutiremos como outras estruturas de dados — tais como *hashing* — podem ser usadas para construir índices. Também apresentaremos brevemente o conceito de índices lógicos, que fornecem um nível adicional, indireto, a partir dos índices físicos, permitindo que um índice físico seja flexível e extensível em sua organização. A Seção 14-6 apresenta um resumo do capítulo.

14.1 TIPOS DE ÍNDECES ORDENADOS EM NÍVEL ÚNICO

A idéia em que se baseia a estrutura de acesso de índice ordenado é a mesma do índice de um livro-texto: listar os termos importantes ao final do livro, em ordem alfabética, com uma lista dos números de páginas onde o termo aparece. Podemos pesquisar um índice para encontrar a lista de *endereços* — números de página, neste caso — e usar esses endereços para localizar um termo no livro-texto, *pesquisando* nas páginas indicadas. A alternativa, se nenhum outro guia nos fosse fornecido, seria

14.1 Tipos de índices Ordenados em Nível Único 327

penetrar lentamente, palavra por palavra, todo o livro-texto para encontrar o termo no qual estivéssemos interessados; isso corresponde a fazer uma busca linear em um arquivo. É claro que a maioria dos livros possui informações adicionais, tais como títulos de capítulo e de seção, que nos ajudam a encontrar um termo sem a necessidade de pesquisar o livro inteiro. Entretanto, o índice é a única indicação exata de onde cada termo se encontra no livro.

Para um arquivo com uma dada estrutura de registros, composto de diversos campos (ou atributos), uma estrutura de acesso de índice geralmente é definida com base em um único campo do arquivo, chamado **campo de indexação** (ou **atributo de indexação**). De maneira geral, o índice armazena cada valor do campo de indexação com uma lista de ponteiros para todos os blocos de disco que contêm registros com aquele valor de campo. Os valores no índice são *ordenados* de forma que possamos realizar uma busca binária. O arquivo de índice é muito menor que o arquivo de dados, assim, a busca do índice por meio da busca binária é razoavelmente eficiente. Os índices multiníveis (Seção 14.2) nos desobrigam da necessidade de realizar a busca binária ao custo da criação de índices para o próprio índice.

Há vários tipos de índices ordenados. Um **índice primário** é especificado sobre o *campo-chave de classificação* de um arquivo ordenado de registros. Relembre a Seção 13.7, na qual um campo-chave de classificação é usado para *ordenar fisicamente* os registros do arquivo no disco, e cada registro possui um *único valor* para o campo. Se o campo de classificação não é um campo-chave — ou seja, se vários registros no arquivo podem ter o mesmo valor para o campo de classificação —, outro tipo de índice, chamado **índice clustering**, pode ser utilizado. Observe que um arquivo pode ter, no máximo, um campo de classificação física, portanto, ele só terá um índice primário ou um índice *clustering*, mas não ambos. Um terceiro tipo de índice, chamado **índice secundário**, pode ser especificado sobre qualquer campo que *não* seja o de *classificação* de um arquivo. Um arquivo pode ter diversos índices secundários além de seu método de acesso primário. Nas próximas três subseções, discutiremos esses três tipos de índices em nível único.

14.1.1 Índices Primários

Um **índice primário** é um arquivo ordenado cujos registros são de tamanho fixo e contêm dois campos. O primeiro campo é do mesmo tipo de dados do campo-chave de classificação — chamado **chave primária** — do arquivo de dados, e o segundo campo é um ponteiro para um bloco de disco (um endereço de bloco). Há uma **entrada de índice** (ou **registro de índice**) no arquivo de índice para cada *bloco* do arquivo de dados. Cada entrada de índice possui, como valores para seus dois campos, o valor do campo-chave primário para o *primeiro* registro em um bloco e um ponteiro para aquele bloco. Nos referiremos aos valores desses dois campos da entrada de índice por $\langle K(i), P(i) \rangle$.

Para criar um índice primário para o arquivo mostrado na Figura 13.7 usamos o campo NOME como chave primária, porque ele é o campo-chave de classificação do arquivo (supondo que cada valor de NOME seja único). Cada entrada no índice possui um valor de NOME e um ponteiro. As primeiras três entradas de índice são as seguintes:

$\langle K(1) = (\text{Aaron, Ed}), P(1) = \text{endereço do bloco 1} \rangle$ $\langle K(2) = (\text{Adams, John}), P(2) = \text{endereço do bloco 2} \rangle$ $\langle K(3) = (\text{Alexander, Ed}), P(3) = \text{endereço do bloco 3} \rangle$

A Figura 14.1 ilustra esse índice primário. O número total de entradas no índice é o mesmo *número de blocos de disco* do arquivo de dados ordenado. O primeiro registro em cada bloco do arquivo de dados é chamado **registro âncora** do bloco, ou simplesmente **âncora do bloco**.

Os índices também podem ser caracterizados como densos ou esparsos. Um **índice denso** possui uma entrada de índice para *cada valor da chave de busca* (portanto, para cada registro) do arquivo de dados. Porém, um **índice esparso** (ou **não-denso**) possui entradas de índice para apenas alguns dos valores de busca. Um índice primário é, portanto, um índice esparso, uma vez que inclui uma entrada de índice para cada bloco do arquivo de dados e as chaves de seus registros âncoras em vez de uma entrada para cada valor de busca (ou cada registro).

O arquivo de índice para um índice primário precisa de muito menos blocos do que o arquivo de dados por duas razões. Primeiro, há menos *entradas de índice* que registros no arquivo de dados. Segundo, cada entrada de índice é tipicamente *menor em tamanho* que o registro de dados, porque elas possuem apenas dois campos; conseqüentemente, mais entradas de índice do que registros de dados podem caber em um bloco. Portanto, uma busca binária no arquivo de índice exige menos acessos a blocos do que uma busca binária no arquivo de dados. Referindo-se de novo à Tabela 13.2, observe que a busca binária para

1 Usaremos indistintamente os termos *campo* e *atributo* neste capítulo.

2 Podemos usar um esquema similar ao descrito aqui, com o último registro em cada bloco (em vez do primeiro), como âncora do bloco. Isso melhora levemente a eficiência do algoritmo de busca.

328 Capítulo 14 Estruturas de Indexação de Arquivos

um arquivo de dados ordenado exigiria $\log_2 b$ acessos a blocos. Mas, se o arquivo de índice primário contém b blocos, então localizar um registro com um valor de chave de busca exige uma busca binária naquele índice e o acesso ao bloco contendo aquele registro: um total de $\log_2 b; + 1$ acesso.

ARQUIVO DE ÍNDICE (entradas $\langle K(1), P(1) \rangle$)

VALOR DA CHAVE PRIMÁRIA

DA ÂNCORA PONTEIRO

DO BLOCO BLOCO

Aaron, Ed

Adams, John

Alexander, Ed

Allen, Troy

Anderson, Zach

Arnold, Mack

Wong, James

Wright, Pam

(CAMPO-CHAVE

PRIMÁRIO)

NOME

ARQUIVO DE DADOS

SSN DATANASC CARGO SALÁRIO SEXO

Aaron, Ed

Abbott, Diane

•

Acosta, Marc

Adams, John

Adams, Robin

•

•

Akers, Jan

Alexander, Ed

Alfred, Bob

•

•

Allen, Sam

Allen, Troy

Anders, Keith

•

•

Anderson, Rob

Anderson, Zach

Angeli, Joe

•

Archer, Sue

Arnold, Mack

Arnold, Steven

•

Atkins, Timothy

Wong, James

Wood, Donald

•

•

Woods, Manny

Wright, Pam

Wyatt, Charles

•

Zimmer, Byron

FIGURA 14.1 índice primário para o campo-chave de classificação do arquivo mostrado na Figura 13.7.

Um registro cujo valor de chave primária seja K está em um bloco cujo endereço é $P(i)$, onde $K(i) < K < K(i+1)$.

O i -ésimo bloco no arquivo de dados contém todos os registros desse tipo por causa da ordem física dos registros do arquivo sob o campo-chave primário. Para recuperar um registro, dado o valor K de seu campo-chave primário, realizamos uma busca binária no arquivo de índice para encontrar a entrada de índice i apropriada, e então recuperamos o bloco do arquivo de dados cujo endereço é $P(i)$.

O Exemplo 1 ilustra a economia em acessos a blocos que pode ser obtida quando um índice primário é usado para buscar um registro.

3 Observe que a fórmula acima não estaria correta se os dados fossem ordenados segundo *um campo que não fosse o campo-chave*; nesse caso, o mesmo valor de índice na âncora do bloco poderia ser repetido nos últimos registros do bloco anterior.

14.1 Tipos de índices Ordenados em Nível Único 329

Exemplo 1: Suponha que tenhamos um arquivo ordenado com $r = 30.000$ registros armazenados em um disco com tamanho de bloco $B = 1.024$ bytes. Os registros do arquivo são de tamanho fixo e são *não-espaçados* (*unspanned*), com tamanho de registro $R = 100$ bytes. O fator de divisão em blocos para o arquivo seria $bfr = L(B/R)J = \lfloor (1-024/100)J \rfloor = 10$ registros por bloco. O número de blocos necessários para o arquivo é $b = Rr/bfr = \lfloor (30.000/10) \rfloor = 3.000$ blocos. Uma busca binária no arquivo de dados exigiria aproximadamente $K(\log_2 b) I = I(\log_2 3.000) I = 12$ acessos a blocos.

Agora suponha que o campo-chave de classificação do arquivo tenha o tamanho de $V = 9$ bytes, um ponteiro de bloco $P = 6$ bytes, e que tenhamos construído um índice primário para o arquivo. O tamanho de cada entrada de índice é $R_i = (9 + 6) = 15$ bytes, de forma que o fator de divisão em blocos para o índice é $bfr_i = L(B/R_i)J = \lfloor (1.024/15)J \rfloor = 68$ entradas por bloco. O número total de entradas de índice r_i é igual ao número de blocos do arquivo de dados, que é 3.000 . O número de blocos de índice e , por isso, $b_i = Rr_i/bfr_i = \lfloor (3.000/68) \rfloor = 45$ blocos. Para realizar uma busca binária no arquivo de índice, seriam necessários $I(\log_2 b_i) I = \lfloor (\log_2 45) I \rfloor = 6$ acessos a blocos. Para buscar um registro usando o índice, precisamos de um acesso adicional a bloco do arquivo de dados, o que daria um total de $6 + 1 = 7$ acessos a blocos — uma otimização em relação à busca binária no arquivo de dados, que exigiria 12 acessos a blocos.

O maior problema com um índice primário — bem como com qualquer outro arquivo ordenado — é a inclusão e a exclusão de registros. Com um índice primário, o problema é composto porque, se tentarmos incluir um registro em sua posição correta no arquivo de dados, devemos não apenas mover registros para abrir espaço para o novo registro, mas também alterar algumas entradas de índice, uma vez que mover os registros irá alterar os registros-âncora de alguns blocos. Usar um arquivo desordenado de *overflow*, conforme visto na Seção 13.7, pode reduzir esse problema. Uma outra possibilidade é utilizar uma lista encadeada para os registros de *overflow* de cada bloco do arquivo de dados. Isso é similar ao método para tratamento de registros de *overflow* descritos para o *hashing* na Seção 13.8.2. Os registros dentro de cada bloco e sua lista encadeada de *overflow* podem ser ordenados para melhorar o tempo de recuperação. A exclusão de registros é feita por meio do uso de marcadores de exclusão.

14.1.2 índices *Clustering*

Se os registros de um arquivo são fisicamente ordenados segundo um campo que não seja um campo-chave — o qual *não* possua um valor distinto para cada registro —, esse campo é chamado **campo de clustering** (*agrupamento*). Podemos criar um tipo diferente de índice, chamado **índice clustering**, para aumentar a velocidade de recuperação de registros que tenham o mesmo valor para o campo de *clustering*. Esse índice difere do índice primário, que requer que um campo de classificação do arquivo de dados possua um *valor distinto* para cada registro.

Um índice *clustering* é também um arquivo ordenado com dois campos; o primeiro é do mesmo tipo do campo de *clustering* do arquivo de dados e o segundo é um ponteiro de bloco. Há uma entrada no índice *clustering* para cada *valor distinto* do campo de *clustering*, contendo o valor e o ponteiro para o *primeiro bloco* do arquivo de dados que possui um registro com aquele valor para seu campo de *clustering*. Temos um exemplo na Figura 14.2. Observe que a inclusão e a exclusão de registros ainda causam problemas porque os registros de dados estão fisicamente ordenados. Para aliviar o problema da inclusão, é comum reservar um bloco inteiro (ou um conjunto [cluster] de blocos adjacentes) para *cada valor* do campo de *clustering*; todos os registros com aquele valor são dispostos no bloco (ou no grupo [cluster] de blocos), tornando a inclusão e a exclusão relativamente diretas. A Figura 14-3 mostra esse esquema.

Um índice *clustering* é outro exemplo de índice *esparso* porque possui uma entrada para cada *valor distinto* do campo de indexação que não é um campo-chave por definição, possuindo, portanto, valores duplicados em vez de uma entrada para cada registro do arquivo. Há, de um lado, uma semelhança entre as figuras 14.1 e 14.3 e, de outro, entre as figuras 14-1 e 13.11. Um índice é, de algum modo, similar às estruturas de diretório usadas no *hashing* extensível, descritos na Seção 13.8.3. Ambos são pesquisados para encontrar um ponteiro para o bloco de dados, que contém o registro desejado. Uma diferença importante é que uma busca no índice usa os valores do próprio campo de busca, enquanto a busca em um diretório *hash* usa o valor *hash*, que é calculado pela aplicação da função *hash* no campo de busca.

14.1.3 índices Secundários

Um **índice secundário** fornece um meio secundário de acesso a um arquivo para o qual já existe algum acesso primário. O índice secundário pode ser usado sobre um campo que é uma chave candidata e possui um valor único em cada registro, ou um campo que não é chave e que possui valores duplicados. O índice é um arquivo ordenado com dois campos. O primeiro é do mesmo tipo de dados de algum campo que não seja o de *classificação* do arquivo de dados, que é um **campo de indexação**. O segundo é um ponteiro de *bloco* ou um ponteiro de *registro*. Pode existir *muitos* índices secundários (e, assim, muitos campos de indexação) para o mesmo arquivo.

```

ARQUIVO DE DADOS
(CAMPO DE CLUSTERING)
NUWLDEPARTAMENTO  NOME  SSN  CARGO  DATANASC  SALÁRIO
ARQUIVO DE ÍNDICE (entradas <K(i), P(i)>;
VALOR DO CAMPO DE CLUSTERING
1
1
1
2
2
3 3 3
3 3 4 4
5 5 5
5
6 6 6 6
6
8
8
8

```

FIGURA 14.2 Um índice *clustering* para o campo NUM_DEPARTAMENTO, que não é campo-chave de classificação, de um arquivo EMPREGADO.

Primeiro, consideremos uma estrutura de acesso de índice secundário em um campo-chave que possua um *valor distinto* para cada registro. Tal campo às vezes é chamado **chave secundária**. Nesse caso, há uma entrada de índice para *cada registro* do arquivo de dados, que contém o valor da chave secundária no registro e um ponteiro para o bloco no qual o registro está armazenado ou para o próprio registro. Assim, tal índice é **denso**.

Novamente nos referimos aos dois valores dos campos da entrada de índice como <K(i), P(i)>. As entradas são **ordenadas** pelo valor de K(i), de modo que possamos realizar uma busca binária. Como os registros do arquivo de dados *não* estão ordenados fisicamente pelos valores do campo chave secundária, *não podemos* usar âncoras de blocos. É por isso que uma entrada de índice é criada para cada registro do arquivo de dados, em vez de uma entrada para cada bloco, como no caso de índice primário. A Figura 14-4 mostra um índice secundário no qual os ponteiros P(i) nas entradas de índice são *ponteiros de bloco*, e não ponteiros de registros. Uma vez que o bloco apropriado é transferido para a memória principal, pode ser executada uma busca do registro desejado dentro do bloco.

Um índice secundário geralmente precisa de mais espaço de armazenamento e de maior tempo de busca que um índice primário, em vista de seu maior número de entradas. Entretanto, a *melhoria* no tempo de busca para um registro arbitrário é muito maior em um índice secundário que num índice primário, uma vez que teria de ser realizada uma *busca linear* no arquivo de dados se o índice secundário não existisse. Para um índice primário, ainda poderíamos usar uma busca binária no arquivo principal, mesmo se o índice não existisse. O Exemplo 2 ilustra a melhoria no número de blocos acessados.

14.1 Tipos de índices Ordenados em Nível Único

(CAMPO DE *CLUSTERING*)

ARQUIVO DE DADOS

NUM3EPARTAMENTO0 NOME SSN CARGO DATANASC SALÁRIO

ARQUIVO DE ÍNDICE (entradas <K(i), P(i)>)

VALOR DO

CAMPO DE PONTEIRO

CLUSTERING DE BLOCO

1

*x

1

' 1

1

ponteiro nulo

2

2

ponteiro nulo

3

3

3

3

ponteiro de bloco •

ponteiro de bloco

ponteiro nulo

4

4

ponteiro de bloco

ponteiro nulo

ponteiro nulo

6

6

6

6

ponteiro de bloco •

• 6

ponteiro de bloco

8

8

8

ponteiro nulo

4. ponteiro nulo

FIGURA 14.3 Um índice *clustering* com um grupo (*cluster*) separado de blocos para cada grupo de registros que compartilhem o mesmo valor de campo de *clustering*.

Exemplo 2: Considere o arquivo do Exemplo 1 com $r = 30.000$ registros de tamanho fixo, $R = 100$ bytes armazenados em um disco com tamanho de bloco $B = 1.024$ bytes. O arquivo possui $b = 3.000$ blocos, conforme calculado no Exemplo 1. Para realizar uma busca linear no arquivo, seriam necessários $b/2 = 3.000/2 = 1.500$ acessos a bloco em média. Suponha que tenhamos construído um índice secundário baseado em um campo do arquivo que não seja o campo-chave de classificação e cujo tamanho seja $V = 9$ bytes. Como no Exemplo 1, um ponteiro de bloco é do tamanho $P = 6$ bytes, assim, o tamanho de

cada entrada de índice é $R_i = (9 + 6) = 15$ bytes, e o fator de divisão em blocos para o índice é $bfr_i = L(B/R_i)J = L(1.024/15)J = 68$ entradas por bloco. Em um índice secundário denso como esse, o número total de entradas de índice $^{\wedge}$ é igual ao *número de registros* do arquivo de dados, que é 30.000. O número de blocos necessários para o índice é, por isso, $b_i = \lceil (r/bfr_i - j) \rceil = \lceil (30.000/68) \rceil = 442$ blocos.

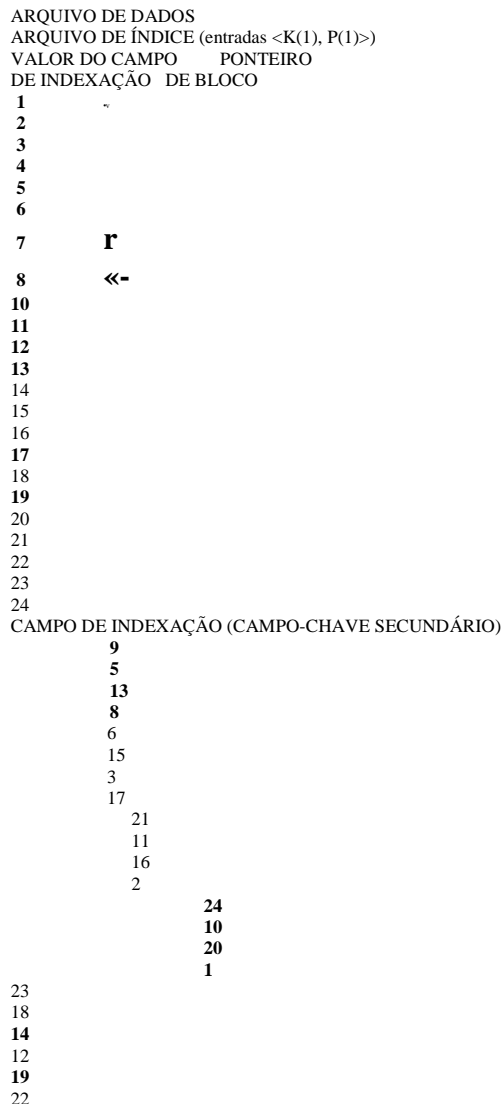


FIGURA 14.4 Um índice secundário denso (com ponteiros de bloco) em um campo que não é chave de classificação de um arquivo.

Uma busca binária neste índice secundário exige $\lceil (\log_2 b_i) \rceil = \lceil \log_2 442 \rceil = 9$ acessos a blocos. Para buscar um registro usando o índice, precisamos de um acesso adicional a bloco do arquivo de dados, o que daria um total de $9 + 1 = 10$ acessos a blocos — uma grande melhoria em relação aos 1.500 acessos a blocos necessários, em média, na busca linear, mas um pouco pior que os sete blocos necessários no índice primário.

Podemos também criar um índice secundário em um campo que *não seja campo-chave* do arquivo. Nesse caso, diversos registros do arquivo podem ter o mesmo valor no campo de indexação. Há diversas opções de implementação para tal índice:

- A Opção 1 consiste em incluir diversas entradas de índice com o mesmo valor $K(i)$ — uma para cada registro. Esse seria um índice denso.
- A Opção 2 consiste em ter registros de tamanho variável nas entradas do índice, com um campo multivalorado para o ponteiro. Mantemos uma lista de ponteiros $\langle P(i,1), \dots, P(i,k) \rangle$ na entrada de índice para $K(i)$ — um ponteiro para cada bloco que contém um registro cujo valor do campo de indexação seja igual a $K(i)$. Em ambas as opções, o algoritmo de busca binária para o índice deve ser alterado adequadamente.

14.1 Tipos de índices Ordenados em Nível Único 333

A Opção 3, que é a mais usada, consiste em manter as próprias entradas de índices em um tamanho fixo e ter uma entrada única para cada valor *do campo de indexação*, porém criando um nível adicional, indireto, a fim de tratar os múltiplos ponteiros. Nesse esquema esparsa, o ponteiro P(i) na entrada de índice <K(i), P(i)> aponta para um *bloco de ponteiros de registro*; cada ponteiro de registro, naquele bloco, aponta para um dos registros do arquivo de dados com valor K(i) no campo de indexação. Se algum valor K(i) ocorrer em muitos registros, de forma que seus ponteiros não caibam em um único bloco de disco, é utilizado um conjunto (*cluster*) ou uma lista encadeada de blocos. Essa técnica é ilustrada na Figura 14.5. A recuperação por meio do índice exige um ou mais acessos adicionais a blocos por causa do nível adicional, mas os algoritmos de busca no índice e (o que é mais relevante) para a inclusão de novos registros no arquivo de dados são diretos. Além disso, as recuperações baseadas em condições complexas de seleção podem ser tratadas por meio de referências aos ponteiros de registros, sem ter de recuperar muitos registros desnecessários do arquivo (Exercício 14.19).

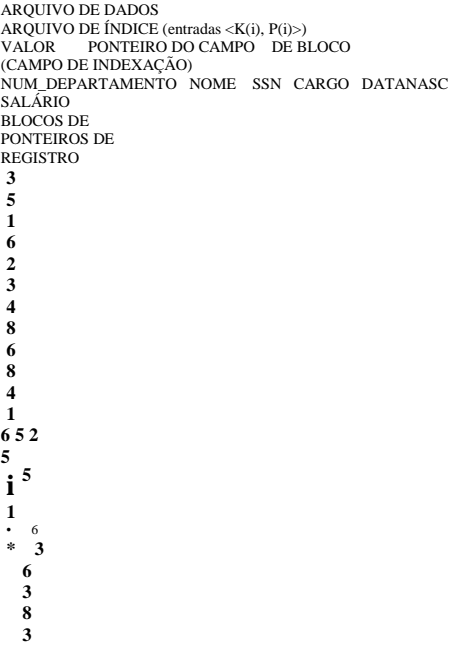


FIGURA 14.5 Um índice secundário (com ponteiros de registro), em um campo que não é campo-chave, implementado em um nível adicional, indireto, de forma que as entradas de índice sejam de tamanho fixo e possuam valores de campo únicos.

Observe que o índice secundário fornece uma **ordem lógica** para os registros, baseada no campo de indexação. Se acessarmos os registros na ordem das entradas no índice secundário, obteremos os mesmos na ordem do campo de indexação.

14.1.4 Resumo

Para concluir esta seção, resumimos a discussão sobre os tipos de índices em duas tabelas. A Tabela 14-1 mostra as características do campo de indexação para cada tipo de índice ordenado em nível único — primário, *clustering* e secundário. A Tabela 14-2 resume as propriedades de cada tipo de índice, comparando o número de entradas de índice e especificando quais são densos e quais usam âncoras de bloco do arquivo de dados.

Campo de Indexação Utilizado	Campo de Indexação Não Utilizado Para para Classificar o Arquivo	Classificar o Arquivo
Campo de indexação é chave indexação não é chave	índice primário índice <i>clustering</i>	índice secundário (campo-chave) índice secundário (campo que não é chave)
TABELA 14.2 Propriedades dos tipos de		índices

Tipo de índice		
Número de Entradas de índice (Primeiro Nível)		
Denso ou		
Esperso		
Âncora de Bloco		
no		
Arquivo de Dados		
Primário		
Clustering		
Secundário (campo-chave)		
Secundário (campo que não é chave)		
Número de blocos do arquivo de dados	Esperso	
Número de valores distintos do campo de indexação	Esperso	Número de registros do arquivo de dados
Sim		Denso
Sim/não	Não	
Número de registros ou número de valores distintos	Denso ou esperso	Não do campo de indexação'
' Sim, se todo valor distinto do campo de indexação iniciar um novo bloco; caso contrário, não. " Para a opção 1. Para as opções 2 e 3.		

14.2 ÍNDICES MULTINÍVEIS

Os esquemas de indexação que descrevemos até aqui envolvem um arquivo de índice ordenado. Uma busca binária é aplicada ao índice para localizar os ponteiros para um bloco do disco ou para um registro (ou registros) do arquivo que possua um valor específico no campo de indexação. Uma busca binária exige aproximadamente (k^b;) acessos a blocos em um índice com bj blocos, porque cada passo do algoritmo reduz à metade a parte do arquivo de índice na qual devemos prosseguir a busca. Essa é a razão pela qual usamos a função log com base 2. A idéia de um **índice multinível** é reduzir a parte do arquivo de índice na qual devemos prosseguir a busca dividindo-a por bfrj, fator de divisão em blocos para o índice, que é maior que 2. Por isso, o espaço de busca é reduzido muito mais rapidamente . O valor bfrj é chamado de *fan-out* do índice multinível, e nos referimos a ele pelo símbolo **fo**. A busca em índice multinível requer aproximadamente (log_{fo}bj) acessos a blocos, que é um número menor que o da busca binária se o *fan-out* for maior que 2.

Um índice multinível leva em conta que o arquivo de índice, ao qual agora passaremos a nos referir como o **primeiro nível** (ou **base**) de um índice multinível, é um *arquivo ordenado* com um *valor distinto* para cada K(i). Por isso, podemos criar um índice primário para o primeiro nível; esse índice para o primeiro nível é chamado **segundo nível** do índice multinível. Como o segundo nível é um índice primário, podemos usar âncoras de bloco de forma que o segundo nível possua uma entrada para *cada bloco* do primeiro nível. O fator de divisão em blocos bfrj, para o segundo nível — e para todos os níveis subsequentes — é o mesmo do índice de primeiro nível porque todas as entradas de índice têm o mesmo tamanho; cada uma tem um valor de campo e um endereço de bloco. Se o primeiro nível tiver rj entradas e o fator de divisão em blocos — que também é o *fan-out* — para o índice for bfrj = fo, então o primeiro nível necessita de I (rj/fo) I blocos, o que, portanto, é o número de entradas r2 necessárias no segundo nível do índice.

Podemos repetir esse processo para o segundo nível. O **terceiro nível**, que é um índice primário para o segundo nível, possui uma entrada para cada bloco do segundo nível, assim, o número de entradas no terceiro nível é r3 = I (r2/fo) |. Observe que necessitamos de um segundo nível apenas se o primeiro nível precisar de mais de um bloco de armazenamento em disco, e, **de** forma similar, de um terceiro nível apenas se o segundo nível precisar de mais de um bloco. Podemos repetir o processo anterior até que todas as entradas de algum nível de índice t caibam em um único bloco. Esse bloco no *t-ésimo* nível é chamado de nível de índice de **topo**. Cada nível reduz o número de entradas do nível anterior dividindo-o por *fo* — o índice de

4 O esquema de numeração para os níveis de índice usado aqui é oposto à forma com que os níveis normalmente são definidos em estruturas de dados de árvores. Nessas estruturas, t é chamado de nível 0 (zero), t - 1 de nível 1 etc.

44 41

51

44

46

51
52

55
58

55
63

63

66

71

71

80

78

80
82

85

85

FIGURA 14.6 Um índice primário de dois níveis que se parece com a organização ISAM (*Indexed Sequential Access Method*—Método de Acesso Sequencial Indexado).

Exemplo 3: Suponha que o índice secundário denso do Exemplo 2 seja convertido em um índice multinível. Calculamos que $b_{fij} = 68$ entradas de índice por bloco, que também é *ofan-out* f_o do índice multinível; também calculamos que o número de blocos de primeiro nível é $b_j = 442$ blocos. O número de blocos de segundo nível será $b_2 = r(b_1/f_o)l = r(442/68)l = 7$ blocos, e o número de blocos do terceiro nível será $b_3 = Kb_2/f_o$ $l = [(7/68)l = 1$ bloco. Por isso, o terceiro nível é o nível de topo do índice, e $t = 3$. Para acessar um registro por meio da busca no índice multinível, devemos acessar um bloco em cada nível, mais um bloco no arquivo de dados, portanto precisamos $det+1=3 + 1=4$ acessos a blocos. Compare com o Exemplo 2, no qual dez acessos a blocos eram necessários quando um índice em nível único e a busca binária foram usados.

Observe que poderíamos também ter um índice primário multinível, que poderia ser esparso. O Exercício 14.14c ilustra este caso, no qual *devemos* acessar o bloco de dados do arquivo antes que possamos determinar se o registro que está sendo procurado está no arquivo. Para um índice denso, isso pode ser determinado acessando-se o primeiro nível de índice (sem necessitar acessar um bloco de dados), uma vez que há uma entrada de índice para *cada* registro do arquivo.

Uma organização de arquivo comum utilizada em aplicações comerciais de processamento de dados é um arquivo ordenado com um índice primário multinível baseado em seu campo-chave de classificação. Tal organização é chamada **arquivo seqüencial ordenado** e foi usada em grande número dos antigos sistemas IBM. A inclusão é tratada por alguma forma de arquivo de *overflow*, incorporado periodicamente ao arquivo de dados. O índice é recriado durante a reorganização. A organização ISAM da IBM incorpora um índice de dois níveis, que é muito parecido com a organização do disco. O primeiro nível é um índice de cilindro, que tem o valor da chave de um registro âncora para cada cilindro do conjunto de disco (*disk pack*), e um ponteiro para o índice de trilha do cilindro. O índice de trilha possui o valor chave de um registro âncora para cada trilha do cilindro e um ponteiro para a trilha. A trilha pode, então, ser pesquisada seqüencialmente para localizar o registro ou o bloco desejado.

O Algoritmo 14.1 esboça o procedimento de busca por um registro em um arquivo de dados que utiliza um índice binário esparso multinível com t níveis. Referimo-nos à entrada i no nível j do índice por $\langle K(i), P(i) \rangle$, e buscamos um registro cujo valor da chave primária é K . Supomos que quaisquer registros de *overflow* são ignorados. Se o registro estiver no arquivo, deve existir alguma entrada no nível 1 tal que $K_1(i) < K < K_1(i+1)$, e o registro estará no bloco do arquivo de d cujo endereço é $P(i)$. O Exercício 14.19 discute a modificação do algoritmo de busca para outros tipos de índice.

Algoritmo 14.1: Busca em um índice primário multinível esparso com t níveis.

```
p <- endereço do bloco do nível de topo do índice;
for j <- t step - 1 to 1 do
begin
  read o bloco do índice (no j-ésimo nível de índice) cujo endereço é p;
  buscar, no bloco p, a entrada i tal que  $C(z) \neq K$ ,  $K_j(i) + 1$  (se  $K_j(i)$  for a última entrada do bloco,
  suficiente satisfazer  $K_j(i) \neq K$ );
  p <-  $P_j(i)$  (* obter o ponteiro apropriado no j-ésimo nível de índice *)
end;
read o bloco do arquivo de dados cujo endereço é p;
buscar, no bloco p, o registro com chave = K;
```

Conforme vimos, um índice multinível reduz o número de acessos a blocos quando pesquisa um registro, dado seu índice no campo de indexação. Ainda enfrentaremos problemas com o tratamento de inclusões e exclusões porque todos os níveis do índice são *arquivos fisicamente ordenados*. Para manter os benefícios do uso de um índice multinível e, ao mesmo tempo reduzir os problemas com a inclusão e exclusão no índice, projetistas adotaram um índice multinível que deixa algum espaço em seus blocos para a inclusão de novas entradas. Trata-se do **índice multinível dinâmico** que freqüentemente é implementado por meio do uso de estruturas de dados chamadas árvores-B e árvores-B⁺, que descreveremos na próxima seção.

14.3 ÍNDICES MULTINÍVEIS DINÂMICOS QUE USAM ÁRVORES-B (B-TREES) e ÁRVORES-B⁺ (B⁺-TREES)

Árvores-B e árvores-B⁺ são casos especiais das bem conhecidas estruturas de dados de árvores. Apresentamos muito brevemente a terminologia utilizada na discussão de estruturas de dados de árvores. Uma **árvore** é formada por **nós** (ou **nodos**). Cada nó em uma árvore, exceto um nó especial chamado **raiz**, tem um nó **pai** e vários (zero ou mais) nós **filhos**. O nó raiz possui um pai. Um nó que não possua nós filhos é chamado nó **folha**; um nó que não é folha é chamado nó **interno**. O **nível** de um nó é sempre uma unidade maior que o nível de seu pai, com o nível do nó raiz sendo igual a zero. Uma **subárvore** de nó consiste daquele nó e de todos os seus nós **descendentes** — seus nós filhos, os nós filhos de seus nós filhos, e assim por diante. Uma definição recursiva precisa de subárvore é que ela consiste de um nó n e das subárvores de todos os nós filhos; a Figura 14-7 ilustra uma estrutura de dados de árvore. Nessa figura, A é o nó raiz, e seus nós filhos são B, C e D. Os nós E, G, H e K são nós folhas.

5 Essa definição padrão para o nível de um nó de árvore, que usamos ao longo desta seção, é diferente daquele que demos para os índices multiníveis na Seção 14.2.

14.3 Índices Multiníveis Dinâmicos que Usam Árvores-B (*B-Trees*) e Árvores-B⁺ (*B⁺-trees*) 337

(os nós E, J, C, G, H e K são nós folhas da árvore)

FIGURA 14.7 Estrutura de dados que mostra uma árvore não-balanceada.

Geralmente mostramos uma árvore com o nó raiz no topo, conforme mostrado na Figura 14.7. Uma maneira de implementar uma árvore é ter tantos ponteiros em cada nó quanto o número de nós filhos daquele nó. Em alguns casos, um ponteiro para o pai também é armazenado em cada nó. Além dos ponteiros, um nó geralmente contém algum tipo de informação armazenada. Quando um índice multinível é implementado usando uma estrutura de árvore, essa informação inclui os valores do campo de indexação do arquivo, que são usados para guiar a busca por um registro em particular.

Na Seção 14.3.1 apresentaremos árvores de busca e, então, veremos as árvores-B, que podem ser usadas como índices multinível dinâmicos para guiar a busca de registros em um arquivo de dados. Os nós de uma árvore-B são preenchidos entre 50% e 100%, e ponteiros para os blocos de dados são armazenados em ambos os nós internos e nós folhas da estrutura da árvore-B. Na Seção 14-3.2 analisaremos as árvores-B⁺, uma variação das árvores-B na qual os ponteiros para os blocos de dados são armazenados apenas nos nós folhas; isso pode levar a menos níveis e a índices de capacidade mais elevada.

14.3.1 Árvores de Busca e Árvores-B

Uma árvore de busca é um tipo especial de árvore, usada para guiar a busca por um registro, dado o valor de um dos campos do registro. Os índices multinível discutidos na Seção 14.2 podem ser vistos como uma variação de uma árvore de busca; cada nó do índice multinível pode ter até *fo* ponteiros e *fo* valores de chave, nos quais *fo* é o *fan-out* do índice. Os valores do campo do índice em cada nó nos guiam para o próximo nó, até que alcancemos o bloco do arquivo de dados que contém os registros requeridos. Ao seguirmos um ponteiro, restringimos nossa busca em cada nível a uma subárvore da árvore de busca e ignoramos todos os nós não pertencentes a essa subárvore.

Árvores de Busca. Uma árvore de busca é um pouco diferente de um índice multinível. Uma árvore de busca de ordem *p* é uma árvore tal que cada nó contenha *pelo menos* *p* - 1 valores de busca e *p* ponteiros na ordem $\langle P_1, K_1, P_2, K_2, \dots, P_j, K_j, P_{q+1} \rangle$, onde $q < p$; cada *P_i* é um ponteiro para um nó filho (ou um ponteiro nulo); e cada *K_j* é um valor de busca para algum conjunto ordenado de valores. Supõe-se que todos os valores de busca sejam únicos. A Figura 14-8 ilustra um nó em uma árvore de busca. Duas condições sempre devem ser satisfeitas em uma árvore de busca:

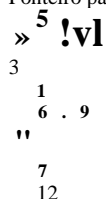
1. Dentro de cada nó, $K_1 < K_2 < \dots < K_{q+1}$.
2. Para todos os valores *X* na subárvore apontada por *P_i*, temos $K_{j-1} < X < K_j$ para $1 < i < q$; $X < K_1$ para $i = 1$; e $K_{q+1} < X$ para $i = q$ (Figura 14.8).

Sempre que buscamos um valor *X*, seguimos o ponteiro *P_i*; apropriado de acordo com as fórmulas da segunda condição apresentada acima. A Figura 14.9 ilustra a árvore de busca de ordem *p* = 3 e de valores de busca inteiros. Observe que alguns ponteiros *P_i* em um nó podem ser ponteiros nulos.

6 Essa restrição pode ser relaxada. Se o índice é baseado em um campo que não é chave, valores duplicados de busca podem existir, e a estrutura do nó e as regras de navegação na árvore podem ser alteradas.

338 Capítulo 14 Estruturas de Indexação de Arquivos

$K,$
 K_{q-1}
 $P,$
 $X < K,$ $K_{i-1} < X < K_i$ $K_{q-1} < X$
 FIGURA 14.8 Um nó em uma árvore de busca com ponteiros para as subárvores abaixo dela.
 Ponteiro para nó de árvore Ponteiro de árvore nulo

FIGURA 14.9 Uma árvore de busca de ordem $p = 3$.

Podemos usar uma árvore de busca como um mecanismo para a busca de registros armazenados em um arquivo em disco. Os valores na árvore podem ser de um dos campos do arquivo, chamado campo de busca (que é o mesmo campo de indexação se um índice multinível guiar a busca). Cada valor de chave na árvore é associado a um ponteiro para o registro de arquivo de dados que possua aquele valor. De maneira alternativa, o ponteiro poderia apontar para o bloco de disco que contém aquele registro. A própria árvore de busca pode ser armazenada em disco, designando cada nó da árvore a um bloco (disco). Quando um novo registro é incluído, devemos atualizar a árvore de busca inserindo, na árvore, uma entrada que contenha o valor do campo de busca do novo registro e um ponteiro para o novo registro.

São necessários algoritmos para a inclusão e para a exclusão de valores de busca na árvore de busca que mantenham duas condições apresentadas anteriormente. Em geral, esses algoritmos não garantem que a árvore de busca seja balanceada, o que significa que todos os seus nós folhas estão em um mesmo nível. A árvore da Figura 14.7 não é balanceada porque possui nós folhas nos níveis 1, 2 e 3. Manter a árvore de busca balanceada é importante porque garante que nenhum nó estará a níveis muito altos e, assim, exigindo muitos acessos a blocos durante uma busca na árvore. Manter a árvore balanceada proporciona uma velocidade de busca uniforme independente do valor da chave de busca. Um outro problema com as árvores de busca é que a exclusão de registros pode deixar alguns nós da árvore quase vazios, desperdiçando espaço de armazenamento e aumentando o número de níveis. As árvores-B tratam ambos os problemas, especificando condições adicionais às das árvores de busca.

Árvores-B. As árvores-B possuem condições adicionais que garantem que uma árvore esteja sempre balanceada e que o espaço desperdiçado pela exclusão, se houver, nunca se torne excessivo. Todavia, os algoritmos para inclusão e exclusão se tornam mais complexos a fim de atender a essas condições. No entanto, a maioria das operações de inclusões e exclusões é simples; elas se tornam complicadas apenas sob circunstâncias especiais — ou seja, sempre que tentarmos uma inclusão em um nó que já estiver completo ou a exclusão de um nó tal que ele fique com menos da metade de sua capacidade preenchida. Mais formalmente, uma árvore-B de ordem p , quando utilizada como uma estrutura de acesso em um *campo-chave* para buscar registros em um arquivo de dados, pode ser definida da seguinte forma:

1. Cada nó interno da árvore-B (Figura 14-10a) é:

$\langle ?_p, \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$

7 A definição de *balanceada* é diferente para as árvores binárias. Árvores binárias balanceadas são conhecidas como árvores AVL.

14.3 Índices Multíniveis Dinâmicos que Usam Árvores-B (*B-Trees*) e Árvores-B⁺ (*B⁺-trees*) 339

onde $q < p$. Cada P_i é um ponteiro de árvore — um ponteiro para outro nó da árvore-B. Cada Pr_i é um **ponteiro de dados** — um ponteiro para o registro cujo valor do campo-chave de busca é igual a K_j (ou para o bloco do arquivo de dados que contém aquele registro).

(a)
 ponteiro de árvore
 $X < K_i$
 K_i $?^r <$
 ponteiro de dados "
 K_{i-1}
 ponteiro para nó de árvore
 ponteiro de dados
 $Pr_{i-1} \mid \mid P_i \mid K_i \mid Pr_i$
 I ponteiro
 I de árvore $\setminus r$
 ponteiro de dados
 $K_M < X < K_i$
 de dados
 $K_{q-1} < X$
 (b)
 1 o 3 o
 0 5 0 1 8 0 n

|
 6 0 M^0

9 0 12 0

Ponteiro para nó de árvore Ponteiro de dados Ponteiro de árvore nulo

FIGURA 14.10 Estruturas de árvore-B. (a) Um nó em uma árvore-B com $q - 1$ valor de busca, (b) Uma árvore-B de ordem $p = 3$. Os valores foram incluídos na seqüência 8, 5, 1, 7, 3, 12, 9, 6.

2. Dentro de cada nodo, $K_j < K_2 < \dots < K_{q-1}$.
 3. Para todos os valores X do campo-chave de busca na subárvore apontada por P_i ; (a i -ésima subárvore, Figura 14.10a), temos: $K_{j-1} < X < K_i$ para $1 < i < q$; $X < K_i$ para $i = 1$; e $K^{q-1} < X$ para $i = q$.
 4. Cada nó possui no máximo p ponteiros de árvore.
 5. Cada nó, exceto a raiz e os nós folhas, possui pelo menos $\lceil p/2 \rceil$ ponteiros de árvore. O nó raiz possui pelo menos dois ponteiros de árvore, exceto se ele for o único nó da árvore.
 6. Um nó com q ponteiros de árvore, $q < p$, possui $q - 1$ valores de campo de chave de busca (e, por isso, tem $q - 1$ ponteiro de dados).
 7. Todos os nós folhas estão no mesmo nível. Os nós folhas possuem a mesma estrutura dos nós internos, exceto que todos os seus ponteiros de árvore P_i são nulos.
- A Figura 14.10b ilustra uma árvore-B de ordem $p = 3$. Observe que todos os valores de busca K na árvore-B são únicos porque supomos que a árvore é usada como uma estrutura de acesso para um campo-chave. Se usarmos uma árvore-B em um *campo que não é campo-chave*, devemos alterar a definição dos ponteiros de arquivo Pr_i para apontar para um bloco — ou conjunto (*cluster*) de blocos — que contenha os ponteiros para os registros do arquivo. Esse nível adicional, indireto, é similar à Opção 3 para índices secundários vista na Seção 14-1.3.
- Uma árvore-B inicia com um único nó raiz (que também é um nó folha) no nível 0 (zero). Uma vez que o nó raiz esteja completo com $p - 1$ valor de chave de busca, e tentarmos incluir uma nova entrada na árvore, o nó raiz se divide em dois nós no nível 1. Apenas o valor do meio é mantido no nó raiz, e os valores restantes são divididos igualmente entre os outros dois nós. Quando um nó, que não é o nó raiz, estiver completo e uma nova entrada for incluída, ele é dividido em dois nós no mesmo nível, e a entrada do meio é transferida para o nó pai com dois ponteiros para os novos nós oriundos da divisão. Se o nó pai estiver cheio, ele também se dividirá. A divisão pode ser propagada até o nó raiz, criando um novo nível se a raiz for dividida. Não analisaremos em detalhe os algoritmos para árvores-B aqui; em vez disso, esboçaremos os procedimentos de busca e inclusão para árvores-B na próxima seção.
- Um ponteiro de dados é um endereço de bloco ou um endereço de registro; o último é essencialmente um endereço de bloco e um deslocamento de registro dentro do bloco.

Se a exclusão de um valor fizer com que um nó seja preenchido com menos da metade de sua capacidade, ele é combinado com seus nós vizinhos, e isso também pode ser propagado até a raiz. Por isso, a exclusão pode reduzir o número de níveis da árvore. Foi demonstrado, por meio de análise e simulação que, após várias inclusões e exclusões aleatórias em uma árvore os nós ficam com aproximadamente 69% de sua capacidade preenchida quando o número de valores na árvore se estabiliza. Isso também ocorre com árvores-B. Se isso acontecer, a divisão e a combinação de nós ocorrerão apenas raramente, de modo que a inclusão e a exclusão se tornarão bastante eficientes. Se o número de valores aumentar, a árvore irá se expandir sem problemas — embora a divisão dos nós possa ocorrer, algumas inclusões irão demorar mais. O Exemplo 4 ilustra como calcularmos a ordem p de uma árvore-B armazenada em disco.

Exemplo 4: Suponha que o campo de busca seja de tamanho $V = 9$ bytes, o tamanho do bloco de disco seja $B = 512$ bytes, um ponteiro de registro (de dados) seja $P_r = 7$ bytes, e um ponteiro de bloco seja $P = 6$ bytes. Cada nó da árvore-B possui no *máximo* p ponteiros de árvore, $p - 1$ ponteiros de dados e $p - 1$ valores de campo-chave de busca (Figura 14.10a), que devem caber e um único bloco de disco se cada nó da árvore-B corresponder a um bloco de disco. Por isso devemos ter:

$$(p \cdot P) + ((p-1) \cdot (P_r + V)) < B$$

$$(p \cdot 6) + ((p-1) \cdot (7 + 9)) < 512$$

$$(22 \cdot p) < 528$$

Podemos escolher p de forma que seja um valor grande, tal que satisfaça a desigualdade acima, que dá $p = 23$ ($p = 24$ n; é escolhido por causa das razões expostas a seguir).

Em geral, um nó de árvore-B pode conter informação adicional necessária para os algoritmos que manipulam a árvore tais como o número de entradas q no nó e um ponteiro para o nó pai. Por isso, antes de realizarmos o cálculo de p descrito anteriormente, devemos reduzir o tamanho do bloco pelo tamanho do espaço necessário para armazenar todas essas informações. A seguir ilustramos como calcular o número de blocos e de níveis para uma árvore-B.

Exemplo 5: Suponha que o campo de busca do Exemplo 4 não seja o campo-chave de classificação e que construamos uma árvore-B neste campo. Suponha que cada nó da árvore-B está com 69% de sua capacidade preenchida. Cada nó, em média, tem $p \cdot 0,69 = 23 \cdot 0,69$, ou aproximadamente 16 ponteiros e, por isso, 15 valores de campo-chave de busca. *Ofan-out médio* fo 16. Podemos começar pela raiz e verificar quantos valores e ponteiros podem existir, em média, em cada nível subsequente:

Raiz:	1 nó	15 entradas	16 ponteiros
Nível 1:	16 nós	240 entradas	256 ponteiros
Nível 2:	256 nós	3.840 entradas	4.096 ponteiros
Nível 3:	4.096 nós	61.440 entradas	

Em cada nível, calculamos o número de entradas multiplicando o número total de ponteiros do nível anterior por 15, número médio de entradas em cada nó. Por isso, para o tamanho de bloco, tamanho de ponteiro e tamanho de campo-chave de busca de dados, uma árvore-B de dois níveis possui, em média, $3.840 + 240 + 15 = 4.095$ entradas; uma árvore-B de três níveis possui, em média, 65.535 entradas.

Algumas vezes as árvores-B são utilizadas como organizações primárias de arquivo. Nesse caso, todos os registros são armazenados dentro dos nós da árvore-B em vez de apenas as entradas <chave de busca, ponteiro de registro>. Isso funciona bem para arquivos com um *número* relativamente *pequeno de registros*, e um *pequeno tamanho de registro*. Caso contrário, o *fan-out* e o número de níveis se tornam muito grandes para permitir um acesso eficiente.

Em resumo, as árvores-B fornecem uma estrutura de acesso multinível, que é uma estrutura de árvore balanceada na qual cada nó está pelo menos preenchido até a metade de sua capacidade. Cada nó em uma árvore-B de ordem p pode ter no máximo $p - 1$ valores de busca.

14.3.2 Árvores-B⁺

A maioria das implementações de índice multinível usa uma variação da estrutura de dados de árvore-B chamada **árvore-B⁺**. Em uma árvore-B, cada valor do campo de busca aparece uma vez em algum nível da árvore, com um ponteiro de dados. Em uma árvore-B⁺ os ponteiros de dados são armazenados *apenas nos nós folhas* da árvore, por isso a estrutura dos nós folhas diferem da estrutura dos nós internos. Os nós folhas têm uma entrada para *cada* valor do campo de busca, com um ponteiro de dados para o registro (ou para o bloco que contém o registro), se o campo de busca for um campo-chave. Para um campo de busca que não seja um campo-chave, o ponteiro aponta para o bloco que contém ponteiros para os registros do arquivo de dado criando um nível adicional indireto.

14.3 Índices Multiníveis Dinâmicos que Usam Árvores-B (*B-Trees*) e Árvores-B⁺ (B⁺-Trees) 341

Os nós folhas de uma árvore-B⁺ são geralmente ligados entre si para proporcionar acesso ordenado aos registros a partir do campo de busca. Esses nós folhas são similares ao primeiro nível (base) de um índice. Os nós internos de uma árvore-B⁺ correspondem aos outros níveis de um índice multinível. Alguns valores do campo de busca dos nós folhas são *repetidos* nos nós internos da árvore-B⁺ a fim de guiar a busca. A estrutura dos *nós internos* de uma árvore-B⁺ de ordem p (Figura 14.11a) é a seguinte:

1. Cada nó interno é da forma:
 $\langle P_i \rangle K_i, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$
 onde $q < p$ e cada P_i é um **ponteiro de árvore**.
2. Dentro de cada nó interno, $K_j < K_2 < \dots < K_j$.
3. Para todos os valores X do campo de busca na subárvore apontada por P_i , temos: $K_{j-1} < X < K_j$, para $1 < i < q$; $X < K_j$ para $i = 1$; e $K^A < X$ para $i = q$ (Figura 14.11a).

(a)

K_i

Ponteiro de árvore

Ponteiro de árvore

^{q-1}

Ponteiro de árvore

X < K_i

(b)

$K_i \quad P' \quad \cdot \quad P' \quad K_j \quad = r \quad K_{q-1} \quad P_r \quad \cdot$

• >

_i >*

« >'

_i _{q-i} próximo

'

r

r

ponteiro para o próximo nó folha da árvore

ponteiro ponteiro de dados de dados

ponteiro de dados

ponteiro de dados

FIGURA 14.11 Os nós de uma árvore-B*. (a) Nó interno de uma árvore-B* com $q - 1$ valores de busca, (b) Nó folha de uma árvore-B⁺ com $q - 1$ valores de busca e $q - 1$ ponteiros de dados.

4. Cada nó interno possui no máximo p ponteiros de árvore.
 5. Cada nó interno possui pelo menos $\lceil p/2 \rceil$ ponteiros de árvore. O nó raiz possui pelo menos dois ponteiros de árvore se ele é um nó interno.
 6. Um nó interno com q ponteiros, $q < p$, possui $q - 1$ valores de campo de busca.
- A estrutura dos *nós folhas* de uma árvore-B⁺ de ordem p (Figura 14-11b) é a seguinte:

1. Cada nó folha é da forma:

$\langle K_1, P_1, \langle K_2, P_2 \rangle, \dots, \langle K_{q-1}, P_{q-1} \rangle, P_{\text{próximo}} \rangle$

onde $q < p$, cada P_i é um ponteiro de dados, e $P_{\text{próximo}}$ aponta para o próximo *nó folha* da árvore-B⁺.

2. Dentro de cada nó folha, $K_1 < K_2 < \dots < K_{q-1}$, $q < p$.

3. Cada P_i é um **ponteiro de dados** que aponta para o registro cujo valor de campo de busca é K_i , ou para um bloco do arquivo que contenha o registro (ou para um bloco de ponteiros de registro que apontam para os registros cujo valor de campo de busca é K_i , se o campo de busca não é uma chave).

4. Cada nó folha possui pelo menos $\lceil p/2 \rceil$ valores.

5. Todos os nós folhas estão no mesmo nível.

Os ponteiros dos nós internos são *ponteiros de árvore* para os blocos que são nós de árvore, enquanto os ponteiros dos nós folhas são ponteiros *de dados* para os registros ou blocos do arquivo de dados — exceto o ponteiro $P_{\text{próximo}}$, que é um ponteiro de

9 Nossa definição segue a de Knuth (1973). Pode-se definir de maneira diferente uma árvore-B* trocando os símbolos $< e < (K_{q-1} < X < K; X < K^A < X)$, mas os princípios continuam os mesmos.

342 Capítulo 14 Estruturas de Indexação de Arquivos

árvore para o próximo nó folha. Iniciando no nó folha mais à esquerda, é possível percorrer os nós folhas tal como em uma lista encadeada, usando os ponteiros $P_{\text{próximo}}$. Isso proporciona um acesso ordenado aos registros de dados no campo de indexação. Um ponteiro P_{anterior} também pode ser incluído. Para uma árvore-B de um campo que não seja campo-chave, um nível adicional, indireto, similar ao mostrado na Figura 14.5, é necessário, assim, os ponteiros P_r são ponteiros de bloco para os blocos que contenham um conjunto de ponteiros de registros para os verdadeiros registros do arquivo de dados, conforme discutido na Opção 3 da Seção 14.1.3.

Como as entradas dos nós internos de uma árvore- B^+ incluem valores de busca e ponteiros de árvore e não têm ponteiros de dados, mais entradas podem ser agrupadas em um nó interno de uma árvore- B^+ que em uma árvore-B similar. Assim, para o mesmo tamanho de bloco (nó), a ordem p será maior na árvore-B que na árvore- B^+ , conforme ilustra o Exemplo 6. Isso pode levar a menos níveis na árvore- B^+ , melhorando o tempo de busca. Como as estruturas dos nós internos e dos nós folhas de uma árvore- B^+ são diferentes, a ordem p pode ser diferente. Usaremos p para denotar a ordem dos nós internos e p_{folha} para denotar a ordem dos nós folhas, que definimos como sendo o número máximo de ponteiros de dados em um nó folha.

Exemplo 6: Para calcular a ordem p de uma árvore-B, suponha que o campo-chave de busca tenha o tamanho $V = 9$ bytes, que o tamanho do bloco seja $B = 512$ bytes, que um ponteiro de registro tenha o tamanho $P_r = 7$ bytes, e que um ponteiro de bloco tenha o tamanho $P = 6$ bytes, tal como no Exemplo 4. Um nó interno da árvore-B pode ter até p ponteiros de árvore e $p - 1$ valores de campo de busca; estes devem caber em um único bloco. Por isso, temos:

$$(p * P) + ((p - 1) * V) < B \quad (p * 6) + ((p - 1) * 9) < 512 \quad (15 * p) < 521$$

Podemos escolher p de forma que seja o maior valor que satisfaça a desigualdade acima, o que dá $p = 34$. Ele é maior que o valor 23 da árvore-B, resultando em uma *fan-out* maior e em mais entradas em cada nó interno de uma árvore-B que na árvore-B correspondente. Os nós folhas da árvore-B terão o mesmo número de valores e ponteiros, exceto pelo fato de os ponteiros serem ponteiros de dados e um ponteiro para o próximo. Por isso, a ordem p_{folha} para os ponteiros folhas pode ser calculada da seguinte forma:

$$(P_{\text{foih}} * (P_r + V)) + P < B \quad (p_{\text{folha}} * (7 + 9)) + 6 < 512 \quad (16 * p_{\text{folha}}) < 506$$

Segue que cada nó folha pode ter até $p_{\text{folha}} = 31$ combinações de valores de chave/ponteiros de dados, supondo que os ponteiros de dados são ponteiros de registro.

Da mesma forma, na árvore-B, podemos necessitar de informação adicional — para implementar os algoritmos de inclusão e exclusão — em cada nó. Essas informações podem incluir o tipo do nó (interno ou folha), o número q de entradas atuais no nó e ponteiros para os nós pai e irmãos. Por isso, antes de realizarmos os cálculos acima para p e para p_{folha} , devemos reduzir o tamanho do bloco pelo tamanho do espaço necessário para todas essas informações. O próximo exemplo ilustra como podemos calcular o número de entradas em uma árvore-B.

Exemplo 7: Suponha que construímos uma árvore-B para o campo do Exemplo 6. Para calcular o número aproximado de entradas da árvore-B, supomos que cada nó da árvore-B está com 69% de sua capacidade preenchida. Em média, cada nó interno terá $34 * 0,69$ ou aproximadamente 23 ponteiros, e por isso, 22 valores. Cada nó folha, em média, terá $0,69 * p_{\text{folha}} = 0,69 * 31$ ou aproximadamente 21 ponteiros de registros de dados. Uma árvore-B terá os seguintes números médios de entradas em cada nível:

Raiz:	1 nó	22 entradas	23 ponteiros
Nível 1:	23 nós	506 entradas	529 ponteiros
Nível 2:	529 nós	11.638 entradas	12.167 ponteiros
Nível folha:	12.167 nós	255.507 entradas	

Para os tamanhos de bloco, de ponteiro e de campo de busca dados acima, uma árvore-B de três níveis mantém, em média, até 255.507 ponteiros de registro. Compare esse resultado com as 65.535 entradas da árvore-B correspondentes do Exemplo 5.

Busca, Inclusão e Exclusão com Árvores- B^+ . O Algoritmo 14.2 esboça o procedimento de busca por um registro utilizando a árvore- B^+ como estrutura de acesso. O Algoritmo 14-3 ilustra o procedimento para a inclusão de um registro

14.3 Índices Multíniveis Dinâmicos que Usam Árvores-B (*B-Trees*) e Árvores-B⁺ (B⁺-Trees) 343

em um arquivo com uma estrutura de acesso de árvore-B. Esses algoritmos supõem a existência de um campo-chave de busca, e eles devem ser modificados adequadamente para o caso de a árvore-B⁺ ser baseada em um campo que não é chave. Ilustraremos a inclusão e a exclusão com um exemplo.

Algoritmo 14-2: Busca por um registro com valor K no campo-chave de busca usando uma árvore-B⁺.

```
n <-r- bloco contendo o nó raiz da árvore-B+; ler o bloco n;
while (n não é um nó folha da árvore-B+) do begin
  q <- número de ponteiros de árvore do nó n;
  if K # n.K, (* n.K, se refere ao i-ésimo valor de campo de busca do nó n *) then n <- n.P, (* n.P, se refere ao i-ésimo ponteiro de
  árvore do nó n *) else if K > n.K,,! then n <- n.Pq else begin buscar, no nó n, uma entrada i tal que n.K,, < K # n.K,,; n <-
  n.P,,; end; ler bloco n end; buscar, no bloco n, uma entrada (K,, Pr,) com K = K,,; (* buscar nó folha *) if encontrado
  then ler o bloco do arquivo de dados cujo endereço é Pr, e recuperar o registro else registro com valor de campo de busca K não
  está no arquivo de dados;
```

Algoritmo 14.3: Inclusão de um registro com valor K no campo chave de busca em uma árvore-B de ordem p.

```
n <- bloco contendo o nó raiz da árvore-B+; ler o bloco n; inicializar a pilha S vazia; while (n não é um nó folha da árvore-B+)
do begin push (incluir) o endereço de n na pilha S;
(* a pilha S mantém os nós pais que são necessários no caso de uma divisão *) q <- número de ponteiros de árvore do nó n;
if K # n.Ki (* n.K, se refere ao i-ésimo valor de campo de busca do nó n *) then n <- n.Pj (* n.P, se refere ao i-ésimo ponteiro
de árvore do nó n *) else if K > n.Kq.i then n <- n.Pq else begin
  buscar, no nó n, uma entrada i tal que n.Kq.j < K # n.K,,; n <-r- n.P,,; end; ler o bloco n end; buscar, no bloco n, uma entrada
(K,, Pr,) com K = K,,; (* buscar nó folha *) if encontrado
then registro já está no arquivo - não pode ser incluído
else (* incluir entrada na árvore-B+ para apontar para o registro *)
begin
  criar entrada (K, Pr) onde Pr aponta para o novo registro;
  if o nó folha n não está cheio
  then incluir entrada (K, Pr) na posição correta no nó n
  else
    begin (* nó folha n está cheio com pfolha ponteiros de registros - é dividido *)
      copiar n em temp (* temp é um nó folha com mais espaço para manter uma entrada adicional *)
      incluir entrada (K, Pr) na posição correta em temp;
      (* temp agora tem pfolha + 1 entradas da forma (K,, Pr,) *) novo <-r- uma nova entrada de nó folha da árvore; novo.Pproximo <-
      n.Pproximo;
```

344 Capítulo 14 Estruturas de Indexação de Arquivos

```

J <- r(pf0, ha + 1) / 21 ;
n <- as primeiras j entradas de temp (até a entrada (Kj Pr^)); n.Pproximo <~
novo; novo <- as entradas restantes de temp; K <- Kj; (* agora devemos
transferir (K, novo) e incluir no nó pai interno - entretanto, se o pai
estiver cheio, a divisão deve se propagar *) terminou <- falso; repeat if
pilha S está vazia
then (* não há nó pai - novo nó raiz é criado para a árvore *) begin
raiz <~ um novo nó interno da árvore, vazio; raiz <~ <n, K, novo>; terminou <-
verdadeiro; end else
begin n <- pop (excluir) da pilha S; if nó interno n não está cheio then begin
(* nó pai não está cheio - não dividir *)
incluir (K, novo) na posição correta no nó interno n; terminou <- verdadeiro
end else
begin (* nó interno n está cheio com p ponteiros de árvore - é dividido *)
copiar n em temp (* temp é um nó interno com mais espaço *); incluir (k, novo)
na posição correta em temp;
(* temp agora possui p+1 ponteiros de árvore *) novo <- um novo nó interno da
árvore, vazio; j ^L((p+ 1)/2)J ; n <- entradas de temp até o ponteiro de
árvore Pd;
(* n contém <Pn, K1, P2, K2....Pj., Kj., Pj> *)
novo <- entradas de temp a partir do ponteiro de árvore Pid;
(* novo contém <Pj+1, Kj+1....Kn., Pp, Kp, Pp+1> *)
K <- Kj;
(* agora devemos transferir (K, novo) e incluir no nó pai interno *) end end
until terminou
end; end;

```

A Figura 14-12 ilustra a inclusão de registros em uma árvore-B⁺ de ordem p = 3 e Pf₀[h_a] = 2- Primeiro, observamos que a raiz é o único nó na árvore, assim ela também é um nó folha. Assim que mais de um nível for criado, a árvore é dividida em nós internos e nós folhas. Observe que *todo valor de chave deve existir em um nível folha*, porque todos os ponteiros de dados estão no nível folha. Entretanto, apenas alguns valores existem nos nós internos para guiar a busca. Observe também que todo valor que aparece em um nó interno também aparece como *o valor mais à direita* em um nível folha de uma subárvore apontada pelo ponteiro de árvore à esquerda do valor.

Quando um *nó folha* está completo e uma nova entrada é incluída nele, ocorre um *overflow* no nó, e ele deve ser dividido. As primeiras j = ⌊((Pfolha + 1)/2)⌋ entradas no nó original são mantidas lá, e as entradas restantes são transferidas para um novo nó folha. O *yésimo* valor de busca é replicado no nó pai interno, e um ponteiro adicional para o novo nó é criado no pai. Estes devem ser incluídos no nó pai em suas seqüências corretas. Se o nó pai interno estiver completo, o novo valor também lhe causará um *overflow*, assim ele deverá ser dividido. As entradas no nó interno até ? — o *yésimo* ponteiro de árvore após a inclusão do novo valor ponteiro, onde j = L((p + 1)/2)J — são mantidas, enquanto o j-ésimo valor de busca é *transferido* para o pai, não replicado. Um nó interno novo irá conter as entradas a partir de P₊ até o final das entradas no nó (Algoritmo 14.3). Essa divisão pode ser propagada para cima até criar um novo nó raiz e um novo nível na árvore-B.

14.3 Índices Multiníveis Dinâmicos que Usam Árvores-B (*B-Trees*) e Árvores-B⁺ (B*-trees) 345

SEQUÊNCIA DE INCLUSÃO: 8, 5, 1, 7, 3, 12, 9, 6

Incluir 1: *overflow* (novo nível)

• j Ponteiro de nó de árvore ~ j Ponteiro de dados Ponteiro de árvore nulo

Incluir 7

Incluir 3: *overflow* (divisão)

LTHIÁS

5 f

"

1 7 |0| 8 |0|

Incluir 12: *overflow*

(divisão, propaga-se novo nível)

• 3 *

Incluir 9

I 1 lofl 3 |o|

| 5 |o|

I lA4 8 |o|

|12|o|

• 5 •

Incluir 6: *overflow* (divisão, propaga-se)

9 0 12 0

FIGURA 14.12 Um exemplo de inclusão em uma árvore-B* de ordem $p = 3$ e $p_{\text{folha}} = 2$.

A Figura 14.13 ilustra uma exclusão em uma árvore-B⁺. Quando uma entrada é excluída, ela sempre é removida do nível folha. Se acontecer de a entrada ocorrer em um nó interno, ela também deverá ser removida de lá. No último caso, o valor à sua esquerda no nó folha deve substituí-la no nó interno, porque aquele valor é agora a entrada mais à direita na subárvore. A exclusão pode causar *underflow* ao reduzir o número de entradas no nó folha abaixo do número mínimo exigido. Nesse caso, tentamos encontrar um nó folha irmão — um nó folha imediatamente à esquerda ou à direita do nó com *underflow* — e redistribuímos as entradas entre o nó e seu irmão, de forma que ambos estejam pelo menos preenchidos até a metade de sua capacidade; caso contrário, o nó é fundido (*merged*) aos seus irmãos, e o número de nós folhas é reduzido. Um método comum

346 Capítulo 14 Estruturas de Indexação de Arquivos

é tentar redistribuir as entradas com o irmão à esquerda; se não for possível, é feita uma tentativa de redistribuir com o irmão à direita. Se isso também não for possível, os três nós são fundidos (*merged*) em dois nós folha. Nesse caso, o *underflow* pode ser propagado para os nós internos porque são necessários um ponteiro de árvore e um valor de busca a menos. Isso pode se propagar e reduzir os três níveis.

SEQÜÊNCIA DE EXCLUSÃO: 5, 12, 9

1 ^ . 6
' I
1 Õj 5 c 6 0

T⁹ • s
w

8 0 9 0 1 [12 0
< 9
' !
8 0 9 0 <

|12 0

7

7 *
Excluir 12: *underflow* (redistribuir)

. 'I' , 6 . .
/ 1 f \
1 Õ , 6 c 1 *

Excluir 9: *underflow* (fundir (com o da esquerda, ainda com *underflow*, reduzir níveis)

7 0 8 0

FIGURA 14.13 Um exemplo de exclusão em uma árvore-B\

Observe que a implementação dos algoritmos de inclusão e exclusão pode exigir ponteiros para o pai e para os irmãos em cada nó, ou o uso de uma pilha como no Algoritmo 14.3. Cada nó também deve incluir o número de entradas que existem nele seu tipo (interno ou folha). Uma outra alternativa é implementar a inclusão e a exclusão como procedimentos recursivos. Variações de árvores-B e árvores-B⁺. Para concluir esta seção, mencionamos brevemente algumas variações de árvore-B e árvore-B⁺. Em alguns casos, a condição 5 da árvore-B (ou da árvore-B⁺), que exige que cada nó esteja

14.4 Índices em Chaves Múltiplas 347

com pelo menos metade de sua capacidade preenchida, pode ser alterada para exigir que cada nó esteja com pelo menos dois terços de sua capacidade preenchida. Nesse caso, a árvore-B é chamada **árvore-B*** (**B*-tree**). Em geral, alguns sistemas permitem que o usuário escolha um **fator de preenchimento** entre 0,5 e 1,0, no qual o último significa que os nós da árvore-B (índice) estão completamente preenchidos. Também é possível especificar dois fatores de preenchimento para uma árvore-B : um para o nível folha e um para os nós internos da árvore. Quando se inicia a construção do índice, cada nó é preenchido até aproximadamente os fatores de preenchimento especificados. Recentemente, pesquisadores têm sugerido relaxar a exigência de que um nó esteja com a metade de sua capacidade preenchida e, em vez disso, permitir que um nó se torne totalmente vazio antes de realizar a fusão (*merging*), a fim de simplificar o algoritmo de exclusão. Estudos de simulação mostram que isso não desperdiça muito espaço adicional sob inclusões e exclusões aleatoriamente distribuídas.

14.4 ÍNDICES EM CHAVES MÚLTIPLAS

Até aqui fizemos a suposição de que as chaves primárias ou secundárias, nas quais os arquivos foram acessados, eram atributos (campos) únicos. Em muitos pedidos de recuperação e atualização, atributos múltiplos estão envolvidos. Se uma certa combinação de atributos for utilizada muito freqüentemente, será vantajoso definir uma estrutura de acesso que forneça acesso eficiente por meio de um valor-chave, que é uma combinação daqueles atributos.

Por exemplo, considere o arquivo EMPREGADO contendo os atributos NRD (número do departamento), IDADE, RUA, CODIGOPOSTAL, SALÁRIO e CODIGO_HABILIDADE, mais a chave SSN (número de seguro social). Considere a consulta: 'Liste os empregados do departamento de número 4 cuja idade é 59'. Observe que ambos NRD e IDADE não são atributos-chave, o que significa que um valor de busca para qualquer um deles irá apontar múltiplos registros. As seguintes estratégias de busca alternativas podem ser consideradas:

1. Supondo que NRD tenha um índice, mas IDADE não, acesse os registros que têm NRD = 4 utilizando o índice, e então selecione entre eles aqueles registros que satisfaçam IDADE = 59.
2. De maneira alternativa, se IDADE é indexada, mas NRD não é, acesse os registros que têm IDADE = 59 utilizando o índice, e então selecione entre eles aqueles registros que satisfaçam NRD = 4.
3. Se os índices foram criados para ambos NRD e IDADE, ambos os índices podem ser usados; cada índice fornece um conjunto de registros ou um conjunto de ponteiros (para blocos ou registros). A interseção desses conjuntos de registros ou de ponteiros resulta naqueles registros que satisfazem ambas as condições, na localização daqueles registros que satisfazem ambas as condições ou nos blocos nos quais os registros satisfazem ambas as condições.

No final, todas essas alternativas dão o resultado correto. Entretanto, se um conjunto de registros que satisfazem cada condição (NRD = 4 e IDADE = 59) individualmente for grande, embora apenas alguns registros satisfaçam a condição composta, então nenhuma das técnicas acima será muito eficiente para o referido pedido de busca. Existe um número de possibilidades para tratar a combinação <NRD, IDADE> ou <IDADE, NRD> como uma chave de busca feita de múltiplos atributos. Esboçamos brevemente essas técnicas abaixo. Referimo-nos às chaves contendo múltiplos atributos, como **chaves compostas**.

14.4.1 índices Ordenados em Atributos Múltiplos

Toda a discussão neste capítulo até aqui ainda se aplica se criarmos um índice em um campo-chave de busca que seja uma combinação de <NRD, IDADE>. A chave de busca é um par de valores <4, 59> no exemplo acima. Em geral, se um índice for criado nos atributos <A₁, A₂, . . . , A_n>, os valores de chave de busca serão tuplas com n valores: <v₁, v₂, . . . , v_n>.

Uma classificação lexicográfica desses valores de tuplas estabelece uma ordem nesta chave de busca composta. Em nosso exemplo, todas as chaves de departamento para o departamento de número 3 precedem aquelas do departamento 4. Assim, <3, n> precede <4, m> para quaisquer valores de m e n. A classificação ascendente das chaves para NRD = 4 seria <4, 18>, <4, 19>, <4, 20>, e assim por diante. A classificação lexicográfica funciona de maneira similar à classificação de cadeias de caracteres. Um índice, em uma chave composta de n atributos, funciona de maneira similar a qualquer índice visto neste capítulo até aqui.

14.4.2 Hashing Particionado

O *hashing* particionado é uma extensão do *hashing* estático externo (Seção 13.8.2), que possibilita o acesso baseado em múltiplas chaves. Ele é adequado apenas para comparações de igualdade; consultas de faixas (*range queries*) não são tratadas. No *hashing* particionado, para uma chave que consiste de n componentes, a função *hash* é projetada para produzir um resultado

348 Capítulo 14 Estruturas de Indexação de Arquivos

com n endereços *hash* separados. O endereço de *bucket* é uma concatenação desses n endereços. Então é possível buscar uma chave de busca composta solicitada, procurando nos *buckets* apropriados que coincidem com as partes do endereço nas quais estamos interessados.

Por exemplo, considere a chave de busca composta <NRD, IDADE>. Se NRD e IDADE são levados a endereços *hash* de 3 bits e de 5 bits, respectivamente, obteremos um endereço de *bucket* de 8 bits. Suponha que NRD = 4 tenha o endereço *hash* '100', e que IDADE = 59 tenha o endereço *hash* '10101'. Então, para buscar pelo valor de busca composto, NRD = 4 e IDADE = 59, deve-se ir ao endereço de *bucket* 100 10101; para buscar todos os empregados com IDADE = 59, serão buscados todos os *buckets* (oito deles) cujos endereços são '000 10101', '001 10101', ... etc. Uma vantagem do *hashing* particionado é que ele pode ser facilmente estendido para qualquer número de atributos. O endereço de *bucket* pode ser projetado de forma que os bits de ordem alta no endereço correspondam aos atributos mais freqüentemente acessados. Além disso, nenhuma estrutura de acesso separada precisa ser mantida para os atributos individuais. A principal desvantagem do *hashing* particionado é que ele não pode tratar consultas de faixas (range queries) em quaisquer dos atributos que o compõem.

14.4.3 Arquivos Grid

Outra alternativa é organizar o arquivo EMPREGADO como um arquivo *grid*. Se desejarmos acessar um segundo arquivo de duas chaves, digamos NRD e IDADE, tal como em nosso exemplo, podemos construir uma matriz com uma escala linear (ou dimensão) para cada um dos atributos de busca. A Figura 14-4 mostra uma matriz para o arquivo EMPREGADO com uma escala linear para NRD e outra para o atributo IDADE. As escalas são definidas de forma que seja obtida uma distribuição uniforme daquele atributo. Assim, em nosso exemplo, mostramos que a escala linear para NRD tem NRD = 1,2 combinados como o valor 0 na escala, enquanto NRD = 5 corresponde ao valor 2 da escala. De maneira similar, IDADE é dividida em uma escala de 0 a 5 por meio do agrupamento de idades, de forma que os empregados sejam distribuídos uniformemente segundo a idade. A matriz mostrada para esse arquivo possui um total de 36 células. Cada célula aponta para algum endereço de *bucket*, no qual os registros correspondentes àquela célula estão armazenados. A Figura 14.4 também mostra a atribuição de células aos *buckets* (apenas parcialmente).

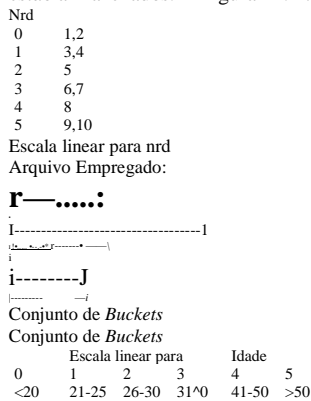


FIGURA 14.14 Exemplo de uma matriz para os atributos NRD e IDADE.

Assim, nossa solicitação por NRD = 4 e IDADE = 59 mapeia a célula (1, 5) de acordo com a matriz. Os registros para essa combinação serão encontrados no *bucket* correspondente. Esse método é particularmente útil para consultas de faixas (range queries) que mapeariam um conjunto de células correspondendo a um grupo de valores ao longo das escalas lineares. Teoricamente, o conceito de arquivo *grid* pode ser aplicado a qualquer número de chaves de busca. Para n chaves de busca, a matriz deve possuir n dimensões. Assim, a matriz permite a partição do arquivo ao longo das dimensões dos atributos-chave de busca e fornece um acesso por meio da combinação de valores naquelas dimensões. Os arquivos *grid* têm um bom desempenho relacionado à redução do tempo para acesso em múltiplas chaves. Entretanto, eles representam uma sobrecarga de espaço relacionado à estrutura de matriz. Além disso, com arquivos dinâmicos, a reorganização freqüente do arquivo aumenta o custo de manutenção.

10 Algoritmos de inclusão/exclusão para arquivos *grid* podem ser encontrados em Nievergelt (1984).

14.5 OUTROS TIPOS DE ÍNDICES

14.5.1 O Uso de *Hashing* Outras Estruturas de Dados como índices

Também é possível criar estruturas de acesso, similares aos índices, que são baseadas em *hashing*. As entradas de índice $\langle K, Pr \rangle$ (ou $\langle K, P \rangle$) podem ser organizadas como um arquivo *hash* dinamicamente extensível, utilizando uma das técnicas descritas na Seção 13.8.3; a busca por uma entrada usa o algoritmo de busca para *hash* baseado em fC. Uma vez que uma entrada é encontrada, o ponteiro Pr (ou P) é utilizado para localizar o registro correspondente no arquivo de dados. Outras estruturas de busca também podem ser utilizadas como índices.

14.5.2 Comparação entre índices Lógicos e Físicos

Até aqui, supomos que as entradas de índice $\langle K, Pr \rangle$ (ou $\langle K, P \rangle$) sempre incluem um ponteiro físico Pr (ou P), que especifica o endereço físico do registro no disco por meio de um número de bloco e um deslocamento. Isso, às vezes, é chamado **endereço físico** e tem a desvantagem de o ponteiro ser alterado se o registro for transferido para uma outra localização no disco. Por exemplo, suponha que uma organização primária de arquivo é baseada em *hashing* linear ou em *hashing* extensível; então, cada vez que um *bucket* for dividido, alguns registros são alocados em novos *buckets* e, por isso, possuem novos endereços físicos. Se havia um índice secundário para o arquivo, os ponteiros para aqueles registros deveriam ser encontrados e atualizados — o que é uma tarefa difícil.

Para melhorar essa situação, podemos usar uma estrutura chamada **índice lógico**, cujas entradas de índice são da forma $\langle K, K_p \rangle$. Cada entrada possui um valor K para o campo de indexação secundária correspondente ao valor K do campo usado na organização primária do arquivo. Por meio de uma busca do valor de K no índice secundário, um programa pode localizar o valor de K_p correspondente e usá-lo para acessar o registro, com o uso da organização primária do arquivo. Assim, os índices lógicos introduzem um nível adicional, indireto, entre a estrutura de acesso e os dados. Eles são utilizados quando se espera que os endereços físicos dos registros mudem freqüentemente. O custo desse nível indireto é uma busca adicional baseada na organização primária do arquivo.

14.5.3 Discussão

Em muitos sistemas, um índice não é uma parte integral do arquivo de dados, mas pode ser criado e descartado dinamicamente. Essa é a razão pela qual, com freqüência, eles são chamados *estrutura de acesso*. Sempre que esperamos acessar freqüentemente um arquivo, a partir de uma condição de busca envolvendo um campo em particular, podemos solicitar ao SGBD a criação de um índice para aquele campo. Geralmente um índice secundário é criado para evitar a classificação física dos registros do arquivo de dados no disco.

A principal vantagem dos índices secundários é que — pelo menos, teoricamente — eles podem ser criados conjuntamente com *quase toda organização primária de registro*. Por isso, um índice secundário poderia ser usado para complementar outros métodos de acesso primário, tais como ordenação ou *hashing*, ou poderia até mesmo ser usado com arquivos mistos. Para criar um índice secundário de árvore-B para algum campo de um arquivo, devemos percorrer todos os registros do arquivo a fim de criar as entradas do nível folha da árvore. Essas entradas são classificadas e preenchidas de acordo com um fator de preenchimento específico; simultaneamente, outros níveis do índice são criados. É mais dispendioso e muito mais difícil criar índices primários e índices *clustering* dinamicamente, porque os registros do arquivo de dados devem ser classificados fisicamente no disco na ordem do campo de indexação. Entretanto, alguns sistemas permitem que os usuários criem esses índices dinamicamente em seus arquivos, por meio da classificação do arquivo, durante a criação do índice.

É comum usar um índice para garantir a *restrição de chave* em um atributo. Enquanto é realizada a busca no índice para incluir um novo registro, pode-se verificar, ao mesmo tempo, se um outro registro do arquivo — daí o porquê da árvore de índice — possui o mesmo valor de atributo-chave do novo registro. Em caso afirmativo, a inclusão pode ser rejeitada.

Um arquivo que possui um índice secundário para cada um dos seus campos é freqüentemente chamado **arquivo completamente invertido**. Como todos os índices são secundários, novos registros são incluídos ao final do arquivo, portanto, o arquivo de dados, propriamente, é um arquivo desordenado (*heap*). Geralmente, os índices são implementados como árvores-B, assim, eles são atualizados dinamicamente para refletir a inclusão ou a exclusão de registros. Alguns SGBDs comerciais, como o ADABAS da Software AG, usam esse método em larga escala.

Fizemos referência à popular organização de arquivo da IBM chamada ISAM na Seção 14.2. Um outro método da IBM, o *Virtual Storage Access Method* (VSAM — Método de Acesso de Armazenamento Virtual), é, de certo modo, similar à estrutura de acesso da árvore-B.

14.6 RESUMO

Neste capítulo apresentamos organizações de arquivo que envolvem estruturas de acesso adicionais, chamadas índices, para aumentar a eficiência da recuperação de registros de um arquivo de dados. Essas estruturas de acesso podem ser usadas *em conjunto* com as organizações primárias de arquivo discutidas no Capítulo 13, que são utilizadas para organizar os próprios registros no arquivo no disco.

Três tipos de índices em nível único foram apresentados: (1) primário, (2) *clustering* e (3) secundário. Cada índice é especificado em um campo do arquivo. Os índices primário e *clustering* são construídos no campo de classificação física do arquivo, enquanto os índices secundários são especificados em campos que não são os de classificação. O campo para o índice primário também deve ser uma chave do arquivo, enquanto para o índice *clustering* não deve ser um campo-chave. Um índice, em nível único, é um arquivo ordenado e é pesquisado por meio do uso de uma busca binária. Mostramos como índices multiníveis podem ser construídos para melhorar a eficiência da busca em um índice.

Depois, mostramos como os índices multiníveis podem ser implementados como árvores-B e árvores-B⁺, as quais são estruturas dinâmicas que permitem que um índice se expanda e encolha dinamicamente. Os nós (blocos) dessas estruturas de índice são mantidos preenchidos entre a metade de sua capacidade até estarem totalmente cheios, por meio de algoritmos de inclusão e exclusão. No final, os nós se estabilizam com uma ocupação média de 69% de sua capacidade, possibilitando espaço para inclusões sem exigir a reorganização do índice na maioria das inclusões. Em geral, as árvores-B⁺ podem manter mais entradas em seus nós internos que as árvores-B, assim, elas podem possuir menos níveis ou manter mais entradas que uma árvore-B correspondente. Fornecemos uma visão geral de métodos de acesso em chaves múltiplas e mostramos como um índice pode ser construído com base em estruturas de dados *hash*. Depois introduzimos o conceito de índice lógico e o comparamos com os índices físicos que descrevemos anteriormente. Finalmente, vimos como combinações das organizações acima podem ser usadas. Por exemplo, os índices secundários são freqüentemente utilizados com arquivos mistos, bem como com arquivos desordenados e ordenados. Os índices secundários também podem ser criados em arquivos *hash* e em arquivos *hash* dinâmicos.

Questões para Revisão

14.1 Defina os seguintes termos: *campo de indexação*, *campo-chave primário*, *campo de clustering*, *campo-chave secundário*, *âncora de bloco*, *índice denso*, *índice esparsa*.

14.2 Quais são as diferenças entre os índices primário, secundário e *clustering*? Como essas diferenças afetam as maneiras pelas quais esses índices são implementados? Quais desses índices são densos e quais não são?

14.3 Por que podemos ter, no máximo, um índice primário ou *clustering* em um arquivo, mas podemos ter diversos índices secundários?

14.4 Como a indexação multinível melhora a eficiência da busca em um arquivo de índice?

14.5 O que é a ordem p de uma árvore-B? Descreva a estrutura dos nós da árvore-B.

14.6 O que é a ordem p de uma árvore-B? Descreva a estrutura de ambos os nós internos e folhas de uma árvore-B. 14-7

Como uma árvore-B se diferencia de uma árvore-B? Por que uma árvore-B é geralmente preferida como uma estrutura de acesso a um arquivo de dados?

14-8 Explique quais as alternativas que existem para acessar arquivos baseados em chaves de busca múltiplas.

14.9 O que é *hashing* particionado? Como funciona? Quais são as suas limitações?

14.10 O que é um arquivo grid? Quais são suas vantagens e suas desvantagens?

14.11 Mostre um exemplo de construção de uma matriz para dois atributos de um arquivo.

14.12 O que é um arquivo completamente invertido? O que é um arquivo seqüencial indexado?

14.13 Como o *hashing* pode ser usado para construir um índice? Qual a diferença entre um índice lógico e um índice físico?

Exercícios

14-14 Considere um disco com tamanho de bloco B = 512 bytes. Um ponteiro de bloco é do tamanho P = 6 bytes, e um ponteiro de registro é do tamanho P_R = 7 bytes. Um arquivo possui r = 30.000 registros de EMPREGADO de tamanho fixo.

14.6 Resumo | 351

Cada registro possui os seguintes campos: NOME (30 bytes), SSN (9 bytes), CODIGODEPARTAMENTO (9 bytes), ENDEREÇO (40 bytes), TELEFONE (9 bytes), DATANASC (8 bytes), SEXO (1 byte), CODIGOCARGO (4 bytes), SALÁRIO (4 bytes, número real). Um byte adicional é usado como marcador de exclusão.

- a. Calcule o tamanho de registro R em bytes.
- b. Calcule o fator de divisão em blocos bfr e o número de blocos de arquivo b , supondo uma organização *não-spanned*.
- c. Suponha que o arquivo é *ordenado* pelo campo-chave SSN e que queiramos construir um *índice primário* para SSN. Calcule (i) o fator de divisão em blocos do índice bfr_i (que também é o *fan-out* fo do índice); (ii) o número de entradas de índice de primeiro nível e o número de blocos de índice de primeiro nível; (iii) o número de níveis necessários se o fizermos em um índice multinível; (iv) o número total de blocos exigidos pelo índice multinível; e (v) o número de acessos a blocos necessários para buscar e recuperar um registro do arquivo — dado seu valor de SSN — utilizando o índice primário.
- d. Suponha que o arquivo *não seja ordenado* pelo campo-chave SSN e que queiramos construir um *índice secundário* para SSN. Repita o exercício anterior (item c) para o índice secundário e compare com o índice primário.
- e. Suponha que o arquivo *não seja ordenado* pelo campo CODIGODEPARTAMENTO, que não é campo-chave, e que queiramos construir um índice secundário para CODIGODEPARTAMENTO utilizando a Opção 3 da Seção 14.1.3, com um nível adicional indireto, que armazena ponteiros de registros. Suponha que existam mil valores distintos de CODIGODEPARTAMENTO e que os registros de EMPREGADO estejam distribuídos igualmente entre esses valores. Calcule (i) o fator de divisão em blocos do índice bfr_i (que também é o *fan-out* fo do índice); (ii) o número de blocos necessários pelo nível indireto que armazena ponteiros de registros; (iii) o número de entradas de índice de primeiro nível e o número de blocos de índice de primeiro nível; (iv) o número de níveis necessários se o fizermos em um índice multinível; (v) o número total de blocos exigidos pelo índice multinível e de blocos utilizados no nível indireto adicional; e (vi) o número aproximado de acessos a blocos necessários para buscar e recuperar todos os registros do arquivo que possuam um valor específico para CODIGODEPARTAMENTO, utilizando o índice.
- f. Suponha que o arquivo *seja ordenado* pelo campo CODIGODEPARTAMENTO, que não é campo-chave, e que queiramos construir um índice *clustering* para CODIGODEPARTAMENTO que use âncoras de bloco (cada novo valor de CODIGODEPARTAMENTO inicia um novo bloco). Suponha que existam mil valores distintos de CODIGODEPARTAMENTO e que os registros de EMPREGADO estejam distribuídos igualmente entre esses valores. Calcule (i) o fator de divisão em blocos do índice bfr_i (que também é o *fan-out* fo do índice); (ii) o número de entradas de índice de primeiro nível e o número de blocos de índice de primeiro nível; (iii) o número de níveis necessários se o fizermos em um índice multinível; (iv) o número total de blocos exigidos pelo índice multinível; e (v) o número de acessos a blocos necessários para buscar e recuperar todos os registros do arquivo que possuam um valor específico para CODIGODEPARTAMENTO, utilizando o índice *clustering* (suponha que múltiplos blocos em um *cluster* sejam adjacentes).
- g. Suponha que o arquivo *não seja ordenado* pelo campo-chave SSN e que queiramos construir uma estrutura de acesso (índice) árvore-B para SSN. Calcule (i) as ordens p e p_{folha} da árvore-B; (ii) o número de blocos de nível de folha necessários se os blocos estiverem com aproximadamente 69% de sua capacidade preenchida (arredondado para cima por conveniência); (iii) o número de níveis necessários se os nós internos também estiverem com aproximadamente 69% de sua capacidade preenchida (arredondado para cima por conveniência); (iv) o número total de blocos exigidos pela árvore-B; e (v) o número de acessos a blocos necessários para buscar e recuperar um registro do arquivo — dado seu valor de SSN — utilizando a árvore-B.
- h. Repita o item g, porém, com árvore-B em vez de árvore-B. Compare seus resultados para a árvore-B com os da árvore-B. 14-15 Um arquivo de PECAS, com NUMERO_PECA como campo-chave, possui registros com os seguintes valores de NUMERO_PECA: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suponha que os valores do campo de busca são incluídos na ordem dada em uma árvore-B de ordem $p = 4$ e $p_{folha} = 3$; mostre como a árvore irá expandir e como a árvore irá ficar no final. 14-16 Repita o Exercício 14-15, mas use uma árvore-B de ordem $p = 4$ em vez de uma árvore-B. 14-17 Suponha que os seguintes valores de campo de busca são excluídos, na ordem dada, da árvore-B do Exercício 14.15; mostre como a árvore irá encolher e como a árvore irá ficar no final. Os valores excluídos são: 65, 75, 43, 18, 20, 92, 59, 37. 14-18 Repita o Exercício 14.17, mas use a árvore-B do Exercício 14-16. 14-19 O Algoritmo 14-1 esboça o procedimento para a busca em um índice primário multinível esparsa para recuperar um registro de arquivo. Adapte o algoritmo para cada um dos seguintes casos:

352

Capítulo 14 Estruturas de Indexação de Arquivos

a. Um índice secundário multinível para um campo que não é chave e que não é campo de classificação do arquivo. Suponha que a Opção 3 da Seção 14-1.3 é utilizada, na qual um nível adicional, indireto, armazena ponteiros para registros individuais com o correspondente valor de campo de indexação.

b. Um índice secundário multinível para um campo-chave, que não é campo de classificação de um arquivo.

c. Um índice *clustering* multinível para um campo de classificação, que não é campo-chave de um arquivo. 14.20 Suponha que existam diversos índices secundários para campos implementados, que não são campo-chave de um arquivo, utilizando a Opção 3 da Seção 14-1.3; por exemplo, poderíamos ter índices secundários para os campos CODIGODEPARTAMENTO, CODIGOCARGO e SALÁRIO do arquivo EMPREGADO do Exercício 14-14. Descreva uma maneira eficiente de buscar e recuperar os registros que satisfaçam uma condição de seleção complexa, baseada nesses campos, tais como (CODIGODEPARTAMENTO = 5 AND CODIGOCARGO = 12 AND SALÁRIO = 50.000), utilizando os ponteiros de registro no nível indireto.

14-21 Adapte para árvores-B os algoritmos 14.2e 14-3, que esboçam os procedimentos de busca e de inclusão em árvores-B.

14.22 É possível modificar o algoritmo de inclusão em árvores-B para atrasar quando um novo nível é gerado, por meio da verificação de uma possível *redistribuição* dos valores entre os nós folhas. A Figura 14.15 ilustra como isso poderia ser feito para nosso exemplo da Figura 14.12; em vez de dividir o nó folha mais à esquerda, quando 12 é incluído, fazemos uma *redistribuição à esquerda*, transferindo 7 para o nó folha à sua esquerda (se houver espaço nesse nó). A Figura 14.15 mostra como a árvore ficaria quando a redistribuição é considerada. Também é possível considerar a *redistribuição à direita*. Tente modificar o algoritmo de inclusão para árvores-B para levar em consideração a redistribuição.

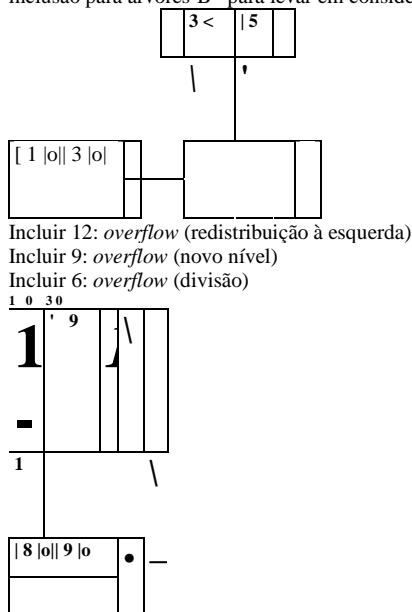


FIGURA 14.15 Inclusão em árvore-B⁺ com redistribuição à esquerda.

14.23 Esboce um algoritmo para a exclusão em uma árvore-B. 14-24 Repita o Exercício 14.23 para uma árvore-B.

14.6 Resumo

353

Bibliografia Seleccionada

Bayer e McCreight (1972) apresentaram as árvores-B e os algoritmos associados. Comer (1979) fornece um excelente levantamento sobre árvores-B e suas origens, e sobre as variações de árvores-B. Knuth (1973) possui uma análise detalhada de muitas técnicas de busca, incluindo árvores-B e algumas de suas variações. Nievergelt (1974) discute o uso de árvores de busca binária para a organização de arquivo. Livros-texto sobre estruturas de dados, entre os quais se incluem Wirth (1972), Claybrook (1983), Smith e Barnes (1987), Miller (1987) e Salzberg (1988), discutem a indexação em detalhes e podem ser consultados a respeito de algoritmos de busca, inclusão e exclusão em árvores-B e árvores-B+. Larson (1981) analisa os arquivos seqüenciais indexados, e Held e Stonebraker (1978) comparam os índices multiníveis estáticos com os índices dinâmicos de árvore-B. Lehman e Yao (1981) e Srinivasan e Carey (1991) realizaram uma análise mais aprofundada sobre o acesso concorrente a árvores-B. Os livros de Wiederhold (1983), Smith e Barnes (1987) e Salzberg (1988), entre outros, discutem muitas das técnicas descritas neste capítulo. Os arquivos *grid* são apresentados por Nievergelt (1984). A recuperação *partialmatch*, que usa *hashing* particionado, é discutida em Burkhard (1976, 1979).

Novas técnicas e aplicações dos índices e das árvores-B são vistas em Lanka e Mays (1991), Zobel *et al.* (1992) e Faloutsos e Jagadish (1992). Mohan e Narang (1992) discutem a criação de índices. O desempenho de vários algoritmos de árvore-B e de árvores-B+ são avaliados em Baeza-Yates e Larson (1989) e Johnson e Shasha (1993). O gerenciamento de *buffer* para índices é discutido em Chan *et al.* (1992).

15

Algoritmos para Processamento e Otimização de Consultas

WÊÊÊÊÊÊk

Neste capítulo veremos as técnicas utilizadas por um SGBD para processar, otimizar e executar consultas de alto nível. Uma consulta expressa em uma linguagem de consulta de alto nível, tal como SQL, deve primeiro passar por uma análise léxica; uma análise sintática e ser validada. A análise léxica (*scanner*) identifica os itens léxicos da linguagem — tais como as palavras-chave da SQL, nomes de atributos e nomes de relacionamentos — no texto da consulta, enquanto a análise sintática (*parser*) verifica a sintaxe da consulta para determinar se ela está formulada de acordo com as regras sintáticas (regras gramaticais) da linguagem de consulta. A consulta também deve ser validada por meio da verificação de que todos os atributos/nomes de relacionamentos são válidos, e se são nomes com significados semânticos no esquema do banco de dados específico que está sendo consultado. Então uma representação interna da consulta é criada, geralmente como uma estrutura de dados de árvore chamada árvore de consulta. Também é possível representar a consulta usando uma estrutura de dados gráfica chamada grafo de consulta. O SGBD, então, deve planejar uma estratégia de execução para a recuperação do resultado da consulta a partir dos arquivos no banco de dados. Em geral, uma consulta possui muitas estratégias de execução possíveis, e o processo de escolha de uma estratégia adequada para o processamento de uma consulta é chamado otimização de consulta.

A Figura 15.1 mostra diferentes passos do processamento de uma consulta de alto nível. O módulo otimizador de consulta tem a função de produzir um plano de execução, e o gerador de código gera o código que executa aquele plano. O processador em tempo de execução do banco de dados tem a função de executar o código da consulta, quer seja no modo interpretado, quer seja no modo compilado, a fim de obter o resultado da consulta. Se resultar em um erro em tempo de execução, uma mensagem de erro é gerada pelo processador em tempo de execução do banco de dados.

O termo *otimização* é, na verdade, uma denominação imprópria porque, em alguns casos, o plano de execução escolhido não é a melhor estratégia (ótima) — ela é apenas uma *estratégia razoavelmente eficiente* para a execução da consulta. Encontrar a estratégia ótima geralmente é uma tarefa que consome muito tempo, exceto para consultas mais simples, e pode exigir informações de como os arquivos são implementados e, até mesmo, do conteúdo dos arquivos — as informações podem não estar completamente disponíveis no catálogo do SGBD. Por isso, *planejamento de uma estratégia de execução* pode ser um denominador mais adequado que *otimização de consulta*.

Para as linguagens de navegação de mais baixo nível de bancos de dados em sistemas legados — tais como as DML de rede ou as HDML hierárquicas (apêndices E e F) —, o programador deve escolher a estratégia de execução de consulta durante a escrita de um programa no banco de dados. Se um SGBD fornece apenas uma linguagem de navegação, há apenas uma *necessidade* ou *oportunidade limitada* para a otimização extensiva da consulta pelo SGBD; em vez disso, ao programador é dada a capacidade de escolher a estratégia de execução 'ótima'. Porém, uma linguagem de consulta de alto nível — como a SQL do '(

¹ Não discutiremos aqui a fase de análise e conferência sintática do processamento de consultas; esse conteúdo é visto em livros-texto de compiladores.

355

SGBDs relacionais (SGBDR) ou OQL (Capítulo 21) dos SGBDs orientados a objeto (SGBDO) — é naturalmente mais declarativa, pois especifica quais são os resultados pretendidos na consulta, em vez de identificar os detalhes de *como* os resultados devem ser obtidos. Portanto, a otimização de consultas é necessária para consultas que são especificadas em uma linguagem de consulta de alto nível.

Consulta em linguagem de alto nível

T

ANÁLISE LÉXICA,
ANÁLISE SINTÁTICA
E VALIDAÇÃO

T

Forma intermediária da consulta

A

OTIMIZADOR DE CONSULTA Plano de execução

1

GERADOR DE CÓDIGO

O código pode ser:

- Executado diretamente (modo interpretado)
- Armazenado e executado posteriormente sempre que necessário (modo compilado)

PROCESSADOR EM TEMPO DE EXECUÇÃO DO BANCO DE DADOS

Resultado da consulta **FIGURA 15.1** Passos típicos durante a execução de uma consulta de alto nível.

Iremos nos concentrar na descrição da otimização de consultas no contexto de um SGBD relacional porque muitas das técnicas que descrevemos foram adaptadas para SGBDs orientados a objeto. Um SGBD relacional deve avaliar sistematicamente estratégias alternativas de execução de consulta e escolher uma estratégia ótima ou razoavelmente eficiente. De maneira geral, cada SGBD possui um número de algoritmos genéricos de acesso ao banco de dados que implementam operações relacionais, tais como SELECT ou JOIN ou combinações dessas operações. Apenas as estratégias de execução que podem ser implementadas pelos algoritmos de acesso do SGBD e que se aplicam a uma consulta específica e a um projeto de banco de dados em particular podem ser consideradas pelo módulo de otimização de consulta.

Iniciaremos na Seção 15.1 com uma discussão geral sobre como as consultas SQL são geralmente traduzidas em consultas de álgebra relacional e, então, otimizadas. Depois, analisaremos os algoritmos para a implementação das operações relacionais da Seção 15.2 até a Seção 15.6. Na sequência, daremos uma visão geral das estratégias de otimização de consultas. Há duas técnicas principais para a implementação da otimização de consultas. A primeira é baseada em **regras heurísticas** para a ordenação das operações em uma estratégia de execução de consulta. A heurística é uma regra que funciona bem na maioria dos casos, mas não se pode garantir que funcionará em todos os casos. Em geral, as regras reordenam as operações em uma árvore de consulta. A segunda técnica envolve a **estimativa** sistemática do custo das diferentes estratégias de execução e da es-

2 Há alguns problemas e algumas técnicas de otimização de consultas pertinentes apenas aos SGBDs orientados a objeto.

Entretanto, não os discutiremos aqui, uma vez que fornecemos apenas uma introdução à otimização de consultas.

U_B UUNSULIA

Código para executar a consulta

colha do plano de execução com o menor custo estimado. Geralmente as duas técnicas são combinadas em um otimizador de consultas. Discutiremos a otimização heurística na Seção 15.7 e a estimativa de custo na Seção 15.8. Depois, na Seção 15.9, forneceremos uma breve visão geral dos fatores considerados durante a otimização de consultas no SGBD relacional comercial ORACLE. A Seção 15.10 apresentará o tópico de otimização semântica de consultas, na qual são utilizadas restrições conhecidas para elaborar estratégias eficientes de execução de consultas.

15.1 TRADUZINDO CONSULTAS SQL PARA A ÁLGEBRA RELACIONAL

Na prática, SQL é a linguagem de consulta utilizada na maioria dos SGBDs comerciais. Primeiro, uma consulta SQL é traduzida em uma expressão equivalente de álgebra relacional estendida — e é representada por uma estrutura de dados de árvore de consulta —, que, então, é otimizada. Em geral, as consultas SQL são decompostas em **blocos de consultas**, que formam as unidades básicas que podem ser traduzidas em operadores algébricos e otimizados. Um bloco de consulta contém uma única expressão SELECT-FROM-WHERE, bem como cláusulas GROUP BY e HAVING, se elas forem partes do bloco. Por isso, consultas aninhadas dentro de uma consulta são identificadas como blocos separados. Como a SQL inclui operadores de agregação — tais como MAX, MIN, SUM e COUNT —, eles devem também ser incluídos na álgebra relacional estendida, conforme vimos na Seção 6.4.

Considere a seguinte consulta SQL na relação EMPREGADO da Figura 5.5:

```
SELECT UNOME, PNAME
FROM EMPREGADO
WHERE SALÁRIO > (SELECT MAX (SALÁRIO)
FROM EMPREGADO
WHERE DN0=5);
```

Essa consulta inclui uma subconsulta aninhada, por isso, seria decomposta em dois blocos. O bloco interno é

```
(SELECT MAX (SALÁRIO) FROM EMPREGADO WHERE DN0=5)
```

e o bloco externo é

```
SELECT UNOME, PNAME FROM EMPREGADO WHERE SALÁRIO > c
```

em que *c* representa o resultado retornado pelo bloco interno. O bloco interno poderia ser traduzido para a seguinte expressão da álgebra relacional estendida

```
^MAX SALÁRIO ("DN0=5"(EMPREGADO))
```

e o bloco externo na seguinte expressão

```
^UNOME.PNAME
(O-SALÁRIO>c(EMPREGADO))
```

O *otimizador de consultas*, então, escolheria um plano de execução para cada bloco. Devemos observar que, no exemplo acima, o bloco interno precisa ser avaliado apenas uma vez para produzir o salário máximo, que depois é utilizado — por meio da constante *c* — pelo bloco externo. Isso foi chamado de *consulta aninhada não-correlacionada* no Capítulo 8. É muito mais difícil otimizar as mais complexas *consultas aninhadas correlacionadas* (Seção 8.5), nas quais uma variável de tupla do bloco externo aparece na cláusula WHERE do bloco interno.

15.2 ALGORITMOS PARA ORDENAÇÃO EXTERNA (*EXTERNAL SORTING*)

A ordenação (*sorting*) é um dos algoritmos primários utilizados no processamento de consultas. Por exemplo, sempre que uma consulta SQL especifica uma cláusula ORDER-BY, o resultado da consulta deve ser ordenado. A ordenação também é um componente-chave nos algoritmos *sort-merge* (ordenação-fusão) usados no JOIN e em outras operações (tais como UNION e INTERSECTION), e em algoritmos de eliminação de duplicatas para a operação PROJECT (quando uma consulta SQL especifica

15.2 Algoritmos para Ordenação Externa (*External Sorting*) 357

a opção DISTINCT na cláusula SELECT). Analisaremos um desses algoritmos nesta seção. Observe que a ordenação pode ser evitada se um índice apropriado existir de forma a permitir o acesso ordenado aos registros.

A **ordenação externa** refere-se aos algoritmos de ordenação que são adequados para arquivos de registros grandes, que são armazenados em disco e que não cabem inteiramente na memória principal, como a maioria dos arquivos de bancos de dados. O típico algoritmo de ordenação externa usa uma **estratégia sort-merge** (ordenação-fusão) que inicia ordenando pequenos subarquivos — chamados **runs** (resultado parcial) — do arquivo principal e, então, realiza a fusão dos **runs** ordenados, criando maiores subarquivos ordenados, que, por sua vez, são fundidos. O algoritmo *sort-merge*, bem como outros algoritmos de bancos de dados, exige *espaço de buffer* na memória principal, onde a ordenação e a fusão dos **runs** são realizadas de fato. O algoritmo básico esboçado na Figura 15.2 consiste de duas fases: (1) a fase da ordenação e (2) a fase de fusão.

inicialize $i \leftarrow -1$;

$k \leftarrow b$; {tamanho do arquivo em blocos} $k \leftarrow n_B$; {tamanho do buffer em blocos} $m \leftarrow r(j/fc)l$; {Fase de Ordenação} while ($i \leq m$) do {

ler os próximos k blocos do arquivo para o buffer ou se houver menos do que k blocos restantes, então ler os blocos restantes; ordenar os registros no buffer e gravá-los como um subarquivo temporário; $i \leftarrow i + 1$;

} {Fase de Fusão: fundir os subarquivos até que reste apenas 1} inicialize $i' \leftarrow 1$;

$p \leftarrow \lfloor \log_{d_M} m \rfloor$; { p é o número de passagens da fase de fusão} $i \leftarrow m$; while ($i \leq p$) do { $n \leftarrow 1$;

$q \leftarrow r/(c-1)l$; {número de subarquivos para gravar nesta passagem} while ($n \leq g$) do { ler os próximos $c-1$ subarquivos ou os subarquivos restantes (da passagem anterior), um bloco por vez; fundir e gravar como novo subarquivo um bloco por vez; $n \leftarrow n + 1$;

; $i' \leftarrow i'$;

$ii \leftarrow ii + 1$;

} **FIGURA 15.2** Esboço de um algoritmo *sort-merge* para ordenação externa.

Na **fase de ordenação**, runs (partes ou pedaços) do arquivo que caibam no espaço de *buffer* disponível são lidos para a memória principal, ordenados usando o algoritmo de classificação interna e escritos de volta no disco como subarquivos (ou *runs*) ordenados temporários. O tamanho de um *run* e o **número inicial de runs** (n_R) são determinados pelo **número de blocos de arquivo** (b) e o **espaço de buffer disponível** (n_B). Por exemplo, se $n_B = 5$ blocos e o tamanho do arquivo for $b = 1.024$ blocos, então $n_R =$

$\lceil b/n_B \rceil$, ou 205 *runs* iniciais, cada um com o tamanho de 5 blocos (exceto o último *run*, que terá 4 blocos). Por isso, após a fase de ordenação, 205 *runs* ordenados estão armazenados como subarquivos temporários no **disco**.

Na **fase de fusão**, os *runs* ordenados são fundidos durante uma ou mais **passagens**. O **grau de fusão** (d_M) é o número de *runs* que podem ser fundidos em cada passagem. Em cada passagem, um bloco de *buffer* é necessário para manter um bloco de cada um dos *runs* que estão sendo fundidos, e um bloco é necessário para conter um bloco de resultado da fusão. Por isso, d_M é o **menor entre** $(n_B - 1) \text{ em } n_R$, e o número de passagens $\lceil \log_{d_M}(n_R) \rceil$. Em nosso exemplo, $d_M = 4$ (fusão de quatro passagens), assim

3 Os algoritmos de ordenação interna (*internal sorting*) são adequados para a ordenação de estruturas de dados que cabem integralmente na memória.

358 Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

os 205 *runs* ordenados iniciais seriam fundidos em 52 no final da primeira passagem, que então são fundidos em 13, depois em 4, depois em 1 *run*, o que significa que *quatro passagens* são necessárias. O d_M mínimo de 2 dá o pior desempenho do algoritmo, que é $(2 * b) + (2 * (b * (\log_2 b)))$

O primeiro termo representa o número de acessos a blocos para a fase de ordenação, pois cada bloco do arquivo é acessado duas vezes — uma vez para a leitura para a memória e uma vez para a escrita dos registros de volta para o disco após a ordenação. O segundo termo representa o número de acessos a blocos para a fase de fusão, supondo o pior caso, d_M de 2. Em geral, o \log é tomado na base d_M e a expressão para o número de acessos a blocos se torna

$$(2 * b) + (2 * (b * (\log_{d_M} n_R)))$$

15.3 ALGORITMOS PARA OPERAÇÕES SELECT E JOIN

15.3.1 Implementação da Operação SELECT

Há muitas opções para a execução da operação SELECT; algumas dependem de o arquivo possuir caminhos específicos de acesso e apenas podem aplicar certos tipos de condições de seleção. Analisaremos alguns desses algoritmos para a implementação de SELECT nesta seção. Usaremos as seguintes operações, especificadas no banco de dados relacional da Figura 5.5, para ilustrar nossa análise:

(OP1): $\sigma_{SSN=123456789}(\text{EMPREGADO})$

(OP2): $\sigma_{DNUMERO \leq 5}(\text{DEPARTAMENTO})$

(OP3): $\sigma_{DNO=5}(\text{EMPREGADO})$

(OP4): $\sigma_{DNO=5 \text{ AND } SALARIO > 30.000 \text{ AND } SEX='F'}(\text{EMPREGADO})$

(OP5): $\sigma_{ESSN=123456789' \text{ AND } PNO=10}(\text{TRABALHA_EM})$

Métodos de Busca para Seleções Simples. Uma variedade de algoritmos de busca é possível para a seleção de registros de um arquivo. Eles também são conhecidos como varreduras de arquivo (*file scans*) porque varrem os registros de um arquivo para buscar e recuperar os registros que satisfazem a condição de seleção. Se o algoritmo de busca envolve o uso de um índice, o índice de busca é chamado de índice de varredura. Os seguintes métodos de busca (de S1 até S6) são exemplos de alguns dos algoritmos de busca que podem ser usados para implementar a operação de seleção:

- S1. *Busca linear (força bruta)*: Recupera *cada registro* do arquivo e testa se seus valores de atributos satisfazem a condição de seleção.
- S2. *Busca binária*: Se a condição de seleção envolver uma comparação de igualdade em um atributo-chave para o qual o arquivo está ordenado, pode-se usar a busca binária — que é mais eficiente que a busca linear. Um exemplo é a OP1 se SSN for o atributo de classificação para o arquivo EMPREGADO.
- S3. *Utilização de um índice primário (ou chave de hash)*: Se a condição de seleção envolver uma comparação de igualdade em um atributo-chave com um índice primário (ou chave de *hash*) — por exemplo, SSN = '123456789' na OP1 —, use o índice primário (ou chave de *hash*) para recuperar o registro. Observe que essa condição recupera (no máximo) um único registro.
- S4. *Utilização de um índice primário para recuperar múltiplos registros*: Se a condição de comparação for $>$, \geq , $<$ ou \leq em um campo-chave com um índice primário — por exemplo, DNUMERO > 5 na OP2 —, use o índice para encontrar o registro que satisfaça a condição de igualdade correspondente (DNUMERO = 5) e depois recupere todos os registros seguintes no arquivo (ordenado). Para a condição DNUMERO < 5 , recupere todos os registros anteriores.

A operação de seleção às vezes é chamada de filtro, uma vez que filtra o arquivo separando os registros que não satisfazem a condição de seleção.

Em geral, a busca binária não é utilizada na busca em bancos de dados porque arquivos ordenados não são usados, a menos que eles também tenham um índice primário correspondente.

1 5.3 Algoritmos para Operações SELECT e JOIN

359

• S5. *Utilização de um índice cluster para recuperar múltiplos registros*: Se a condição de seleção envolver uma comparação de igualdade em um atributo que não seja chave com um índice *clustering* — por exemplo, $DNO = 5$ na OP3 —, use o índice para recuperar todos os registros que satisfaçam a condição.

• S6. *Utilização de um índice secundário (árvore-B) em uma comparação de igualdade*: Este método de busca pode ser usado para recuperar um único registro se o campo de indexação for uma chave (possui valores únicos) ou para recuperar múltiplos registros se o campo de indexação não for chave. Ele também pode ser usado para comparações envolvendo $>$, $>=$, $<$ ou $<=$.

Na Seção 15.8 veremos como desenvolver fórmulas para estimar o custo do acesso nesses métodos de busca em função do número de acessos a blocos e do tempo de acesso. O Método SI se aplica a qualquer arquivo, mas todos os outros métodos dependem da existência de um caminho de acesso apropriado para o atributo utilizado na condição de seleção. Os Métodos S4 e S6 podem ser usados para recuperar registros dentro de uma certa *faixa* — por exemplo, $30.000 \leq SALÁRIO \leq 35.000$. Consultas envolvendo tais condições são chamadas de consultas de faixas (*range queries*).

Métodos de Busca para Seleções Complexas. Se uma condição de uma operação SELECT é uma condição con-juntiva — ou seja, se ela é formada por diversas condições simples conectadas pelo conectivo lógico AND tal como a OP4 acima —, o SGBD pode usar os seguintes métodos adicionais para implementar a operação:

• S7. *Seleção conjuntiva utilizando um índice individual*: Se um atributo envolvido em qualquer condição simples individual da condição conjuntiva possuir um caminho de acesso que permita o uso de um dos métodos de S2 a S6, use aquela condição para recuperar os registros e depois verifique se cada registro recuperado *satisfaz as condições simples restantes* da condição conjuntiva.

• S8. *Seleção conjuntiva utilizando um índice composto*: Se dois ou mais atributos estiverem envolvidos em condições de igualdade na condição conjuntiva e houver um índice composto (ou estrutura *hash*) para a combinação dos campos — por exemplo, se um índice tiver sido criado para a chave composta (ESSN, PNO) do arquivo TRABALHA_EM da OP5 —, podemos usar o índice diretamente.

• S9. *Seleção conjuntiva por meio da intersecção de registros*: Se índices secundários (ou outros caminhos de acesso) estiverem disponíveis para mais de um dos campos envolvidos nas condições simples de uma condição conjuntiva, e se os índices incluem ponteiros de registros (em vez de ponteiros de blocos), cada índice poderá ser usado para recuperar o conjunto de ponteiros de registros que satisfaça a condição individual. A intersecção desses conjuntos de ponteiros de registros resulta nos ponteiros de registros que satisfazem a condição conjuntiva e que são usados depois para recuperar diretamente aqueles registros. Se apenas algumas das condições possuírem índices secundários, cada registro recuperado será posteriormente testado para determinar se ele satisfaz as condições restantes.

Sempre que uma condição individual especifica a seleção — tal como em OP1, OP2 ou OP3 —, podemos apenas verificar se existe um caminho de acesso no atributo envolvido naquela condição. Se houver um caminho de acesso, o método correspondente àquele caminho será utilizado; caso contrário, a abordagem da força bruta da busca linear do método SI será utilizada. A otimização de consulta para uma operação SELECT é necessária principalmente em condições de seleção conjuntivas, sempre que *mais que um* dos atributos envolvidos nas condições possuírem um caminho de acesso. O otimizador deve escolher o caminho de acesso que *recupera o menor número de registros*, de maneira mais eficiente, por meio da estimativa dos diferentes custos (Seção 15.8) e da escolha do método com menor custo estimado.

Quando o otimizador está escolhendo entre múltiplas condições simples em uma condição de seleção conjuntiva, ele considera, em geral, a seletividade de cada condição. A seletividade (s) é definida como sendo a razão do número de registros (tuplas) que satisfazem a condição pelo número total de registros (tuplas) do arquivo (relação) e, assim, é um número entre zero e 1 — a seletividade zero significa que nenhum registro satisfaz a condição e 1 significa que todos os registros satisfazem a condição. Embora as seletividades exatas de todas as condições possam não estar disponíveis, estimativas de seletividade são mantidas freqüentemente no catálogo do SGBD e são utilizadas pelo otimizador. Por exemplo, para uma condição de igualdade em um atributo-chave da relação $r(R)$, $s = 1/I \cdot r(R)$, na qual $I \cdot r(R)$ é o número de tuplas na relação $r(R)$. Para uma condição de igualdade em um atributo com i valores distintos, s pode ser estimado por $(I \cdot r(R) / i) / I \cdot r(R)$, ou $1/i$, supondo que os

6 Um ponteiro de registro identifica exclusivamente um registro e fornece o endereço do registro no disco; por isso ele também é chamado de identificador de registro ou *id* de registro.

7 Uma técnica pode ter diversas variações — por exemplo, os índices poderiam ser *índices lógicos* que armazenam valores da chave primária em vez de ponteiros de registros.

Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

registros são igualmente distribuídos entre os valores distintos. Sob esta suposição, $I r(R) \setminus ji$ registros irão satisfazer a condição de igualdade no atributo. Em geral, o número de registros que satisfazem uma condição de seleção com seletividade s é estimado por $I r(R) I * s$. Quanto menor for essa estimativa, mais desejável é que essa condição seja a primeira a ser utilizada para recuperar os registros.

Em comparação a uma condição de seleção conjuntiva, uma condição disjuntiva (na qual as condições simples são conectadas pelo operador lógico OR em vez de AND) é muito mais difícil de processar e de otimizar. Por exemplo, considere a OP4':

(UP7y): CT1000,5 OR SALARIO>30.000 OR SEXCI='F'(EMPREGADO;

Com uma condição desse tipo, pouca otimização pode ser feita, porque os registros que satisfazem a condição disjuntiva são a *união* dos registros que satisfazem as condições individuais. Por isso, se qualquer *uma* das condições não possuir um caminho de acesso, somos compelidos a usar a abordagem da força bruta da busca linear. Se apenas houver um caminho de acesso para *toda* condição, poderemos otimizar a seleção por meio da recuperação dos registros que satisfazem cada condição — ou seus identificadores de registro —, e então aplicar a operação de união para eliminar duplicidades.

Um SGBD coloca à disposição muitos dos métodos vistos acima e, em geral, outros métodos adicionais. O otimizador de consulta deve escolher o método apropriado para a execução de cada operação SELECT em uma consulta. Essa otimização usa fórmulas que estimam os custos de cada método de acesso disponível, conforme discutiremos na Seção 15.8. O otimizador escolhe o método de acesso com o menor custo estimado.

15.3.2 Implementação da Operação JOIN

A operação JOIN é uma das que mais consome tempo no processamento de consultas. Muitas das operações de junção encontradas nas consultas são variações de operações EQUIJOIN e NATURAL JOIN, por isso consideraremos apenas essas duas. Até o restante deste capítulo, o termo junção (*join*) se referirá a uma EQUIJOIN (ou NATURAL JOIN). Há muitas possibilidades de implementar uma junção de duas vias, que é uma junção em dois arquivos. As junções que envolvem mais de dois arquivos são chamadas **junções de múltiplas vias**. O número de maneiras possíveis para executar as junções de múltiplas vias aumenta muito rapidamente. Nesta seção analisaremos apenas técnicas para a implementação de junções de duas vias. Para ilustrar nossa discussão, faremos referência ao esquema relacional da Figura 5.5 mais uma vez — especificamente, as relações EMPREGADO, DEPARTAMENTO e PROJETO. Os algoritmos que consideramos são para operações de junção da forma

$R \times_{A=B} S$

em que A e B são atributos compatíveis com os domínios de R e S, respectivamente. Os métodos que vimos podem ser estendidos para formas mais genéricas de junção. Ilustramos quatro técnicas mais comuns para a execução desse tipo de junção, utilizando os seguintes exemplos de operações:

(OP6): EMPREGADO DO DNO=DNUMERO DEPARTAMENTO (OP7): DEPARTAMENTO X_{GERSSN=SSN} EMPREGADO

Métodos para a Implementação de Junções (*Joins*):

- J1. *Junção de laços aninhados (nested-loop) (força bruta)*: Para cada registro t em R (laço externo), recupere cada registro s de S (laço interno), e teste se os dois registros satisfazem a condição de junção $t[A] = s[B]$.
- J2. *junção de laço único (single-loop) (utilizando uma estrutura de acesso para recuperar os registros correspondentes à junção)*: Se existir um índice (ou chave de *hash*) para um dos dois atributos da junção — digamos B de S —, recupere cada registro t em R, um por vez (laço único), e, depois, use a estrutura de acesso para recuperar diretamente todos os registros s correspondentes de S que satisfaçam $s[B] = t[A]$.
- J3. *junção sort-merge (ordenação-fusão)*: Se os registros de R e S estiverem *classificados* (ordenados) *fisicamente* pelos valores dos atributos de junção A e B, respectivamente, poderemos implementar a junção da maneira mais eficiente possível. Ambos os arquivos são varridos simultaneamente na ordem dos atributos de junção, fazendo a correspondência dos registros que possuem os mesmos valores para A e B. Se os arquivos não estiverem classifi-

8 Em otimizadores mais sofisticados, histogramas que representam a distribuição dos registros entre os diferentes valores de atributos podem ser mantidos no catálogo.

9 Para arquivos em disco, é óbvio que os laços serão sobre blocos de disco, de forma que essa técnica também foi chamada de *junção de blocos aninhados (nested-block)*.

15.3 Algoritmos para Operações SELECT e JOIN

361

cados, eles deverão ser classificados primeiro por meio de uma ordenação externa (Seção 15.2). Nesse método, pares de blocos de arquivos são ordenadamente copiados para *buffers* de memória, e os registros de cada arquivo são varridos apenas uma vez para realizar a correspondência com o outro arquivo — a menos que ambos, A e B, não sejam atributos-chave e, nesse caso, o método precisa ser levemente modificado. Um esboço do algoritmo de junção *sort-merge* é apresentado na Figura 15.3a. Usamos $R(i)$ para nos referir ao i -ésimo registro em R. Uma variação da junção *sort-merge* pode ser usada quando existirem índices secundários para ambos os atributos de junção. Os índices proporcionam a capacidade de acessar (varrer) os registros na ordem dos atributos de junção, mas os registros de fato estão fisicamente espalhados pelos blocos do arquivo, de forma que esse método pode ser bastante ineficiente, uma vez que cada acesso a registro pode envolver o acesso a um bloco de disco diferente.

```
(a) ordenar as tuplas de R baseando-se no atributo A; (* suponha que R tenha n tuplas (registros) *) ordenar as tuplas de S
baseando-se no atributo B (* suponha que S tenha m tuplas (registros) *) inicializar /←1, y←1; while (h<n) e (j<m) do { if
R(i)[A] > S(j)[B]
then fazer /←/+1 elseif R(i)[A] < S(j)[B] then fazer /←~;+1
else { (* R(i)[A] = S(j)[B], portanto realizamos o output de uma tupla *) output a tupla combinada <R(i), S(j)> em T;
(* output outras tuplas correspondentes a R(i), se houver *) fazer A-y+1;
while (kím) and (R(i)[A] = S(r)[B])
do { output a tupla combinada <R(i), S(i)> em T;
fazer /←/+1;
} (* output outras tuplas correspondentes a S(j), se houver *) fazer jH-H-1;
while (k<n) and R(k)[A] = S(j)[B]
do { output a tupla combinada <R(k), S(j)> em T;
fazer /c←/c+1; } fazer K-H-1, /←/+1 }
criar uma tupla f [ <lista de atributos> ] em T para cada tupla t de R;
(* T contém o resultado da projeção antes da eliminação de duplicatas *) if <lista de atributos> incluir uma chave de R then T<r-T
else { ordenar as tuplas de T inicializar K-1, /←-2; while /← n
do { output a tupla T[i] em T;
while T[i] = T[j] and j<n do /←/+1; (* eliminar duplicatas *) '←/'; y←/+1 } } (* T contém o resultado da projeção após a eliminação
de duplicatas *)
```

FIGURA 15.3 Implementação de JOIN, PROJECT, UNION, INTERSECTION e SET DIFFERENCE por meio de *sort-merge*, quando R possui n tuplas e S possui m tuplas. (a) Implementação da operação $T \leftarrow R \bowtie_{MB} S$. (b) Implementação da operação $T \leftarrow TT_{\langle \text{hsU!kalnbMm} \rangle}(R)$.

• J4- *Junção-hash (hash-join)*: Os registros dos arquivos R e S são particionados (*hashed*) em um mesmo arquivo *hash*, utilizando a mesma função *hash* com os atributos de junção A de R e B de S como chaves de *hash*. Primeiro, uma única passagem pelo arquivo com menor número de registros (digamos, R) coloca seus registros nos *buckets* do arquivo *hash* — essa é a chamada fase de separação, uma vez que os registros de R são separados pelos *buckets* do ar-

Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

quivo *hash*. Então, na segunda fase, chamada de **fase de sondagem**, uma única passagem pelo outro arquivo (S) coloca cada um de seus registros para *investigar* o *bucket* adequado, e aquele registro é combinado com todos os registros correspondentes de R naquele *bucket*. Essa descrição simplificada da junção *hash* supõe que o menor dos dois arquivos *cabe integralmente nos buckets de memória* após a primeira fase. Abaixo veremos variações da junção *hash* que não exigem essa suposição.

Na prática, as técnicas J1 a J4 são implementadas por meio do acesso *integral dos blocos de disco* de um arquivo, em vez de registros individuais. Dependendo da disponibilidade de espaço de *buffer* na memória, o número de blocos lidos do arquivo pode ser ajustado.

(c) ordenar as tuplas de Re S utilizando os mesmos e únicos atributos de ordenação; inicializar w-1; y'<-1;

while (i<ri) and (y'<m) do { if R(i) > S(j)

then { output S(j) em 7; fazer y'<-y+1

} elseif R(i) < S(j) then { output R(i) em 7; fazer i<-i+1

} else fazer y<-y+1 (* R(i)=S(j), portanto, pular uma das tuplas duplicatas *)

} if (i<ri) then adicionar as tuplas a partir de R(i) até R(n) em 7; if (i<m) then adicionar as tuplas a partir de S(j) até S(m) em 7;

(d) ordenar as tuplas de S utilizando os mesmos e únicos atributos de ordenação; inicializar i<-1; y''<-1;

while (i<ri) and (j<m) do { if R(i) > S(j)

then fazer y'<-y+1 elseif R(i) < S(j) then fazer K<-K+1

else { output R(i) em 7; (* R(i)=S(j), portanto, fazemos o output da tupla *) fazer M<-M+1, i'<-i+1 }

(e) ordenar as tuplas de Re S utilizando os mesmos e únicos atributos de ordenação; inicializar i<-1; y<-1;

while (i'<ri) and (i<m) do { if R(i) > S(j)

then fazer y'<-y+1

elseif R(i) < S(j)

then { output R(i) em 7; (* R(i) não tem S(i) correspondente, portanto, fazemos o output de R(i) *) fazer i'<-i+1;

} else fazer i'<-i+1, y<-y+1 } if (i'<ri) then adicionar as tuplas a partir de R(i) até R(n) em 7;

FIGURA 15.3 Implementação de JOIN, PROJECT, UNION, INTERSECTION e SET DIFFERENCE por meio de *sort-merge*,

quando R possui n tuplas e S possui m tuplas. (c) Implementação da operação $R \cup S$. (d) Implementação da operação $R \setminus S$. (e) Implementação da operação $R \cap S$. (continuação)

Efeitos da Disponibilidade de Espaço de Buffer e o Fator de Seleção de Junção no

Desempenho da Junção. O espaço de *buffer* disponível tem um efeito importante nos vários algoritmos de junção.

Primeiro, consideremos a

15.3 Algoritmos para Operações SELECT e JOIN 363

abordagem de laços aninhados (J1). Olhando novamente para a operação OP6, suponha que o número de *buffers* disponíveis na memória principal para a implementação da junção seja $n_B = 7$ blocos (*buffers*). Para ilustração, suponha que o arquivo DEPARTAMENTO consista de $r_D = 50$ registros armazenados em $b_D = 10$ blocos de discos, e que o arquivo EMPREGADO consista de $r_E = 6.000$ registros armazenados em $b_E = 2.000$ blocos de discos. É vantajoso ler para a memória, de uma só vez, quantos blocos forem possíveis do arquivo cujos registros forem utilizados no laço externo (ou seja, $n_B - 2$ blocos). Depois, o algoritmo pode ler um bloco por vez do arquivo do laço interno e usar seus registros para **sondar** (ou seja, pesquisar) os registros correspondentes nos blocos do laço externo na memória. Isso reduz o número total de acessos a blocos. Um bloco de *buffer* adicional é necessário para conter os registros resultantes após sofrerem a junção, e o conteúdo desse bloco de *buffer* é acrescentado ao final do **arquivo de resultado** — o arquivo de disco que contém o resultado da junção — sempre que ele estiver cheio. Então, esse bloco de *buffer* é reutilizado para manter os registros resultantes adicionais.

Na junção de laços aninhados, faz diferença qual dos arquivos é escolhido para o laço externo e para o laço interno. Se o arquivo EMPREGADO for usado para o laço externo, cada bloco de EMPREGADO será lido uma vez, e o arquivo DEPARTAMENTO inteiro (cada um de seus blocos) será lido uma vez a *cada vez* que lermos $(n_B - 2)$ blocos do arquivo EMPREGADO. Temos o seguinte:

Número total de blocos acessados para o arquivo externo = b_E

Número de vezes que $(n_B - 2)$ blocos do arquivo externo são carregados = $\lfloor b_E / (n_B - 2) \rfloor$

Número total de blocos acessados para o arquivo interno = $b_D * \lceil b_E / (n_B - 2) \rceil$

Por isso, temos o seguinte número total de acessos a blocos:

$b_E + (\lfloor b_E / (n_B - 2) \rfloor * b_D) = 2.000 + (\lfloor (2.000/5) \rfloor * 10) = 6.000$ acessos a blocos

Porém, se usarmos os registros de DEPARTAMENTO no laço externo, por simetria, teremos o seguinte número total de acessos a blocos:

$b_D + (\lfloor b_D / (n_B - 2) \rfloor * b_E) = 10 + (\lfloor (10/5) \rfloor * 2.000) = 4.010$ acessos a blocos

O algoritmo de junção usa um *buffer* para manter os registros que sofreram a junção do arquivo de resultado. Uma vez que o *buffer* esteja cheio, ele é escrito no disco e reutilizado. Se o arquivo de resultado de uma operação de junção possuir $\lceil b_{RES} \rceil$ blocos de disco, cada bloco é escrito uma vez, assim, b_{RES} acessos adicionais a blocos devem ser acrescentados às fórmulas anteriores a fim de estimar o custo total da operação de junção. O mesmo ocorre para as fórmulas desenvolvidas posteriormente para os outros algoritmos de junção. Conforme mostra esse exemplo, é vantajoso usar o arquivo com *menor número de blocos*, como o arquivo do laço externo na junção de laços aninhados.

Outro fator que afeta o desempenho de uma junção, particularmente o método de laço único J2, é a porcentagem de registros em um arquivo que sofrerão a junção com os registros do outro arquivo. Chamamos isso de **fator de seleção da junção** de um arquivo em função de uma condição *equijoin* com um outro arquivo. Esse fator depende da condição *equijoin* específica entre os dois arquivos. Para ilustrar isso, considere a operação OP7, a qual faz a junção de cada registro de DEPARTAMENTO com o registro de EMPREGADO para o gerente daquele departamento. Aqui, espera-se que cada registro de DEPARTAMENTO (há 50 desses registros em nosso exemplo) participe da junção com um *único* registro de EMPREGADO, mas muitos registros de EMPREGADO (os 5.950 deles que não gerenciam um departamento) não participarão da junção.

Suponha que índices secundários existam em ambos os atributos SSN de EMPREGADO e GERSSN de DEPARTAMENTO, com o número de níveis de índices $x_{SSN} = 4$ e $x_{GERSSN} = 2$, respectivamente. Temos duas opções para a implementação do método J2. A primeira recupera cada registro de EMPREGADO e depois usa o índice para GERSSN de DEPARTAMENTO para encontrar um registro correspondente em DEPARTAMENTO. Nesse caso, nenhum registro correspondente será encontrado para empregados que não gerenciam um departamento. O número de acessos a blocos aqui é de aproximadamente:

$b_E + (r_E * (x_{GERSSN} + 1)) = 2.000 + (6.000 * 3) = 20.000$ acessos a blocos

A segunda opção recupera cada registro de DEPARTAMENTO e depois usa o índice para SSN de EMPREGADO para encontrar um registro de gerente correspondente em EMPREGADO. Nesse caso, cada registro de DEPARTAMENTO terá um registro correspondente em EMPREGADO. O número de acessos a blocos nesse caso é de aproximadamente:

$$b_D + (r_D * (x_{SSN} + 1)) = 10 + (50 * 5) = 260 \text{ acessos a blocos}$$

10 Se reservarmos dois *buffers* para o arquivo de resultado, o *buffering* duplo pode ser usado para aumentar a velocidade do algoritmo (Seção 13.3).

11 Esse caso é diferente da *seletividade da junção*, que discutiremos na Seção 15.8. 364 Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

A segunda opção é mais eficiente porque o fator de seleção de junção de DEPARTAMENTO em função da condição de junção SSN = GERSSN é 1, enquanto o fator de seleção da junção de EMPREGADO em função da mesma condição de junção é (50/6.000), o 0,008. Para o método J2, tanto o menor arquivo quanto o arquivo que tem uma correspondência para cada registro (ou seja, arquivo com o maior fator de seleção de junção) deveria ser usado no laço (externo) de junção. Também é possível criar um índice especificamente para a execução da operação de junção se um índice ainda não existir.

A junção *sort/merge* J3 é bastante eficiente se ambos os arquivos já estiverem ordenados segundo seus atributos de junção. Apenas uma única passagem é feita em cada arquivo. Por isso, o número de acessos a blocos é igual à soma dos números de blocos de ambos os arquivos. Para esse método, ambas OP6 e OP7 necessitariam de $b_E + b_D = 2.000 + 10 = 2.010$ acessos a blocos. Entretanto, exige-se que ambos os arquivos estejam ordenados segundo os atributos da junção; se um ou ambos não estiverem, eles devem ser ordenados especificamente para a execução da operação de junção. Se estimarmos o custo da ordenação de um arquivo externo em $(b \log_2 b)$ acessos a blocos, e se ambos os arquivos precisarem ser ordenados, o custo total da junção *sort-merge* pode ser estimada em $(b_E + b_D + b_E \log_2 b_E + b_D \log_2 b_D)$.

Junção Hash Particionado e Junção Hash Híbrido. O método de junção *hash* J4 também é bastante eficiente. Nesse caso, apenas uma passagem única é feita em cada arquivo se os arquivos estiverem ordenados ou não. Se a tabela *has* para o menor dos dois arquivos puder ser mantida inteiramente na memória principal após o *hash* (partição), segundo se atributo de junção, a implementação é direta. Entretanto, se partes do arquivo *hash* tiverem de ser ordenadas no disco, o método se torna mais complexo. Diversas variações para melhorar a eficiência têm sido propostas. Analisaremos duas técnicas: junção *hash* particionado e uma variação chamada junção *hash* híbrido, que têm se mostrado bastante eficientes.

No algoritmo da junção *hash* particionado, primeiramente cada arquivo é particionado em M partições utilizando um função *hash* particionado nos atributos da junção. Depois, é feita a junção de cada par de partições. Por exemplo, suponhamos que estejamos realizando a junção das relações R e S nos atributos de junção $R.A$ e $S.B$:

$$R \bowtie_{A=B} S$$

Na fase de partição, R é particionado em M partições R_1, R_2, \dots, R_M , e S em M partições S_1, S_2, \dots, S_M . Uma propriedade de cada par de partições correspondentes R_i e S_i é que os registros de R_i só precisam ter a junção realizada com os registros de S_i , e vice-versa. Essa propriedade é garantida pelo uso da mesma função *hash* para particionar ambos os arquivos em seus atributos de junção — atributo A em R e o atributo B em S . O número mínimo de *buffers* de memória necessários para a fase de partição é $M+1$. Cada um dos arquivos R e S são particionados separadamente. Para cada uma das partições, um único *buffer* de memória — cujo tamanho é um bloco de disco — é alocado para armazenar os registros que são levados pelo *hash* a essa partição. Sempre que um *buffer* de memória para uma partição ficar cheio, seu conteúdo é acrescentado ao final de um subarquivo de disco, que armazena essa partição. A fase de partição possui duas iterações. Após a primeira iteração, o primeiro arquivo R estará particionado nos subarquivos R_1, R_2, \dots, R_M , nos quais todos os registros que sejam levados pelo *hash* a R estarão na mesma partição. Após a segunda iteração, o segundo arquivo S estará particionado de maneira similar.

Na segunda fase, chamada de fase de junção ou de sondagem, M iterações são necessárias. Durante a iteração i , é realizada a junção das duas partições R_i e S_i . O número mínimo de *buffers* necessários para a iteração i é o número de blocos das duas partições, digamos R_i , mais dois *buffers* adicionais. Se usarmos uma junção de laços aninhados durante a iteração i , os registros de R_i da menor das duas partições são copiados nos *buffers* de memória; depois, todos os blocos da outra partição S_i são lidos — um por vez —, e cada registro é utilizado para investigar (ou seja, pesquisar) o(s) registro(s) correspondente(s) na partição R_i . É feita a junção de quaisquer registros correspondentes e estes são escritos no arquivo resultado. Para melhorar a eficiência da sondagem na memória, é comum usar uma *tabela hash na memória* para armazenar os registros da partição R_i por meio do uso de função *hash* diferente da função *hash* particionado.

Podemos calcular aproximadamente o custo dessa junção *hash* particionado como sendo $3 * (b_R + b_S) + b_{RES}$ em nosso exemplo, uma vez que cada registro é lido e escrito de volta no disco uma vez, durante a fase de partição. Durante a fase de junção (sondagem), cada registro é lido uma segunda vez para realizar a junção. A principal dificuldade desse algoritmo é assegurar que a função *hash* particionado seja uniforme — ou seja, que os tamanhos das partições sejam aproximadamente iguais. Se a função de partição é *skewed* (não-uniforme), então algumas partições podem ser muito grandes para caber no espaço da memória disponível para a segunda fase de junção.

12 Podemos usar as fórmulas mais precisas da Seção 15.2 se soubermos o número de *buffers* disponíveis para a ordenação.

13 Se a função *hash* utilizada para a partição for utilizada novamente, todos os registros em uma partição serão levados pelo *hash*, de novo para o mesmo *bucket*.

15.4 Algoritmos para as Operações PROJECT e de CONJUNTO 365

Observe que, se o espaço de *buffer* disponível na memória n_B satisfizer $n_B > (b_R + 2)$, em que b_R é o número de blocos do *menor* dos dois arquivos que participam da junção, digamos R , não há razão para realizar a partição, uma vez que, nesse caso, a junção pode ser realizada inteiramente na memória usando alguma variação da junção de laços aninhados baseada em *hash* e sondagem. Para ilustrar, suponha que estejamos realizando a operação de junção OP6, repetida abaixo:

(OP6): EMPREGADO DO QN0-DNÚMERO DEPARTAMENTO

Nesse exemplo, o menor arquivo é DEPARTAMENTO; por isso, se O número de *buffers* disponíveis na memória n_B satisfizer $n_B > (b_D + 2)$, o arquivo DEPARTAMENTO inteiro pode ser lido para a memória principal e organizado em uma tabela *hash* segundo o atributo da junção. Então, cada bloco de EMPREGADO é lido para um *buffer*, e cada registro de EMPREGADO no *buffer* é levado pelo *hash* segundo seu atributo de junção e é utilizado para *investigar* o correspondente *bucket* na memória da tabela *hash* de DEPARTAMENTO. Se um registro correspondente for encontrado, a junção dos registros é realizada, e o(s) registro(s) resultante(s) é(são) escrito(s) no *buffer* de resultado e finalmente no arquivo de resultado no disco. O custo em função dos acessos a blocos é, por isso, $(b_D + b_E)$, mais b_{RE} — o custo da escrita do arquivo de resultado.

O **algoritmo da junção hash híbrido** é uma variação da junção *hash* particionado, na qual a fase de *junção* para uma das partições é incluída na fase de partição. Para ilustrar, vamos supor que o tamanho de um *buffer* de memória seja um bloco de disco, tal que n_B *buffers* estejam disponíveis, e que a função *hash* utilizada seja $h(K) = K \bmod M$, tal que M partições estejam sendo criadas, onde $M < n_B$. Como exemplo, suponha que estejamos realizando a operação de junção OP6. Na *primeira passagem* da fase de partição, quando o algoritmo de junção *hash* híbrido estiver particionando o menor dos dois arquivos (DEPARTAMENTO em OP6), o algoritmo divide o espaço de *buffer* entre as M partições de tal forma que todos os blocos da *primeira partição* de DEPARTAMENTO residam completamente na memória principal. Para cada uma das outras partições, apenas um único *buffer* em memória — cujo tamanho é um bloco de disco — é alocado; o restante da partição é escrito no disco, bem como na junção *hash* particionada normal. Por isso, ao final da *primeira passagem da fase de partição*, a primeira partição de DEPARTAMENTO residirá inteiramente na memória principal, enquanto cada uma das outras partições de DEPARTAMENTO estarão em um subarquivo no disco.

Na segunda passagem da fase de partição, os registros do segundo arquivo participante da junção — o maior arquivo, EMPREGADO em OP6 — estão sendo particionados. Se um registro for levado pelo *hash* para a *primeira partição*, será feita a sua junção com o registro correspondente em DEPARTAMENTO e os registros resultantes da junção serão escritos no *buffer* de resultado (e eventualmente no disco). Se um registro de EMPREGADO for levado pelo *hash* para uma outra partição, diferente da primeira, ele será particionado normalmente. Por isso, ao final da segunda passagem da fase de partição, todos os registros que são levados pelo *hash* para a primeira partição já terão a sua junção realizada. Agora há $M - 1$ pares de partições no disco. Portanto, durante a segunda fase, de **junção** ou de **sondagem**, são necessárias $M - 1$ iterações em vez de M . O objetivo é realizar a maior quantidade possível de junções de registros durante a fase de partição de forma a economizar o custo de armazenamento desses registros de volta no disco da releitura dos mesmos uma segunda vez durante a fase de junção.

15.4 ALGORITMOS PARA AS OPERAÇÕES PROJECT E DE CONJUNTO

A implementação de uma operação PROJECT (projeção) $\pi_{\langle \text{lista de atributos} \rangle}(R)$ é "direta se a <lista de atributos> incluir uma chave da relação R , porque, nesse caso, o resultado da operação terá o mesmo número de tuplas de R , mas com apenas os valores dos atributos da <lista de atributos> para cada tupla. Se a <lista de atributos> não incluir uma chave de R , *tuplas duplicatas devem ser eliminadas*. Geralmente isso é feito por meio da ordenação do resultado da operação e, depois, pela eliminação de tuplas duplicatas, que aparecem consecutivamente após a ordenação. Um esboço do algoritmo é dado na Figura 15.3b. O *hashing* também pode ser utilizado para eliminar as duplicatas: como cada registro é levado pelo *hash* e incluído em um *bucket* do arquivo *hash* na memória, ele é verificado em relação àqueles que já estão no *bucket*; se for uma duplicata, ele não é incluído. É útil relembrar aqui que, em consultas SQL, o padrão (*default*) é não eliminar as duplicatas do resultado da consulta; apenas se a palavra-chave DISTINCT (distinto) estiver incluída, as duplicatas são eliminadas do resultado da consulta.

As operações de conjunto — UNION (união), INTERSECTION (intersecção), SET DIFFERENCE (diferença de conjuntos) e CARTESIAN PRODUCT (produto cartesiano) — às vezes são dispendiosas para ser implementadas. Em particular, a operação CARTESIAN PRODUCT $R \times S$ é bastante dispendiosa, porque seu resultado inclui um registro para cada combinação de registros de R e S . Além disso, os atributos do resultado incluem todos os atributos de R e S . Se R possuir n registros e j atributos e S possuir m registros e k atributos, a relação resultante terá $n * m$ registros e $j + k$ atributos. Por isso, é importante evitar a operação CARTESIAN PRODUCT e substituí-la por outras operações equivalentes durante a otimização da consulta (Seção 15.7).

As outras três operações de conjuntos — UNION, INTERSECTION e SET DIFFERENCE — se aplicam apenas às relações compatíveis na união, que possuem o mesmo número de atributos e os mesmos domínios de atributos. A maneira costumeira de implementar essas operações é utilizar variações da **técnica sort-merge**: as duas relações são ordenadas segundo o mesmo atributo e, após a ordenação, uma única varredura por meio de cada relação é suficiente para produzir o resultado. Por exemplo, podemos implementar a operação UNION, R U S, por meio da varredura e da fusão de ambos os arquivos ordenados simultaneamente e, sempre que houver a mesma tupla em ambas as relações, apenas uma é mantida no resultado da fusão. Para a operação INTERSECTION, R ∩ S, mantemos no resultado da fusão apenas aquelas tuplas que aparecem em *ambas as relações*. A Figura 15.3, do item (c) ao (e), apresenta esboços da implementação dessas operações por meio da ordenação e da fusão. Alguns dos detalhes não estão incluídos nesses algoritmos.

O **hashing** também pode ser utilizado para implementar UNION, INTERSECTION e SET DIFFERENCE. Uma tabela é particionada e a outra é utilizada para investigar a partição adequada. Por exemplo, para implementar R U S, primeiro aplique o *hash* (particione) aos registros de R; depois, aplique o hash (investigue) aos registros de S, porém, não inclua registros duplicatas nos *buckets*. Para implementar R ∩ S, primeiro particione os registros de R no arquivo *hash*. Depois, enquanto aplica o *hash* em cada registro de S, verifique se um registro idêntico de R é encontrado no *bucket* e, se isso ocorrer, adicione o registro ao arquivo de resultado. Para implementar R - S, primeiro leve os registros de R por meio do *hash* aos *buckets* do arquivo *hash*. Enquanto aplica o *hash* (investiga) em cada registro de S, se um registro idêntico for encontrado no *bucket*, remova o registro do *bucket*.

15.5 IMPLEMENTAÇÃO DAS OPERAÇÕES DE AGREGAÇÃO E OUTER JOINS

15.5.1 Implementação de Operações de Agregação

Os operadores de agregação MIN (mínimo), MAX (máximo), COUNT (contagem), AVERAGE (média), SUM (somatório), quando aplicados a uma tabela inteira, podem ser computados por meio da varredura da tabela ou pelo uso de um índice apropriado se estiver disponível. Por exemplo, considere a seguinte consulta SQL:

```
SELECT MAX(SALARIO) FROM EMPREGADO;
```

Se existir um índice (ascendente) para SALÁRIO na relação EMPREGADO, então o otimizador pode decidir pelo uso do índice para buscar o maior valor seguindo pelo ponteiro *mais à direita* em cada nó do índice, desde a raiz até a folha mais à direita. Aquele nó incluiria o maior valor de SALÁRIO como sua *última* entrada. Na maioria dos casos, isso seria mais eficiente que uma varredura completa da tabela EMPREGADO, uma vez que nenhum registro de fato precisa ser recuperado. A agregação MIN pode ser tratada de maneira similar, exceto que o ponteiro *mais à esquerda* é seguido desde a raiz até a folha mais à esquerda. Esse não possui o menor valor de SALÁRIO como sua *primeira* entrada.

O índice também poderia ser utilizado para as agregações COUNT, AVERAGE e SUM, mas apenas se for um **índice denso** — ou seja, se existir uma entrada de índice para cada registro do arquivo principal. Nesse caso, o cálculo associado seria aplicado aos valores no índice. Para um **índice esparsa**, o número real de entradas associadas a cada entrada de índice deve ser utilizado para corrigir o cálculo (exceto para COUNT DISTINCT [contagem distinta], no qual o número de valores distintos pode ser contado a partir do próprio índice).

Quando a cláusula GROUP BY (agrupar por) é utilizada em uma consulta, o operador de agregação deve ser aplicado separadamente a cada grupo de tuplas. Por isso, primeiro a tabela deve ser particionada em subconjuntos de tuplas, em que cada partição (grupo) possui o mesmo valor para os atributos de agrupamento. Nesse caso, o cálculo é mais complexo. Considere a seguinte consulta:

```
SELECT DNO, AVG(SALARIO) FROM EMPREGADO
GROUP BY DNO;
```

A técnica usual para tais consultas é primeiro utilizar a **ordenação** ou o **hashing**, segundo os atributos de agrupamento para particionar o arquivo nos grupos apropriados. Depois, o algoritmo calcula a função de agregação para as tuplas em cada grupo, as quais possuem o mesmo valor para o(s) atributo(s) de agregação. Na consulta do exemplo, o conjunto de tuplas por departamento seria agrupado em uma partição, e a média de salários seria calculada para cada grupo.

14 SET DIFFERENCE (diferença de conjuntos) é chamada de EXCEPT em SQL.

Observe que, se um **índice clustering** (Capítulo 13) existir para o(s) atributo(s) de agrupamento, então os registros já estarão *particionados* (agrupados) em subconjuntos apropriados. Nesse caso, é necessário apenas aplicar o cálculo em cada grupo.

15.5.2 Implementação do Outer Join

Na Seção 6.4, a *operação de junção externa* (*outer join*) foi apresentada, com suas três variações: junção externa à esquerda ((*e/t outer join*), junção externa à direita (*right outer join*) e junção externa total (*full outer join*). Também discutimos, no Capítulo 8, como essas operações podem ser especificadas na SQL. A seguir temos um exemplo em SQL da operação de junção externa à esquerda:

```
SELECT UNOME, PNAME, DNAME
```

```
FROM (EMPREGADO LEFT OUTER JOIN DEPARTAMENTO ON DNO=DNUMBER);
```

O resultado dessa consulta é uma tabela de nomes de empregados e seus departamentos associados. Esse resultado é similar ao da junção (interna) comum, com exceção de que, se uma tupla de EMPREGADO (uma tupla da relação à esquerda) não tiver um departamento associado, o nome do empregado mesmo assim aparecerá na tabela resultante, mas o nome do departamento seria *nuil*. para tais tuplas no resultado da consulta.

A junção externa pode ser obtida por meio da modificação dos algoritmos de junção, tais como a junção de laços aninhados ou a junção de laço único. Por exemplo, para obter a junção externa à esquerda, usamos a relação à esquerda como laço externo ou laço único porque toda tupla na relação à esquerda deve aparecer no resultado. Se houver tuplas correspondentes na outra relação, as tuplas que participam da junção serão produzidas e salvas no resultado. Entretanto, se nenhuma tupla correspondente for encontrada, a tupla ainda é incluída no resultado, porém, é completada com valor(es) *nuil*. Os algoritmos *sort-merge* e de junção *hash* também podem ser estendidos para gerar as junções externas.

De maneira alternativa, a junção externa pode ser obtida por meio da execução de uma combinação de operadores da álgebra relacional. Por exemplo, a operação de junção externa à esquerda mostrada acima é equivalente à seguinte sequência de operações relacionais:

1. Calcule o JOIN (interno) das tabelas EMPREGADO e DEPARTAMENTO.

```
TEMP1 <- TTUNOME, PNAME, DNAME (EMPREGADO *DNO=DNUMBER DEPARTAMENTO)
```

2. Encontre as tuplas de EMPREGADO que não aparecem no resultado do JOIN (interno).

```
TEMP2 <- TTUNOME, PNAME (EMPREGADO) - TTUNOME, PNAME (TEMP1)
```

3. Complete cada tupla de TEMP2 com valor *nuil* para o campo DNAME.

```
TEMP2 <- TEMP2 X'NULL'
```

4. Aplique a operação UNION em TEMP1 e TEMP2 para produzir o resultado do LEFT OUTER JOIN.

```
RESULTADO <- TEMP1 U TEMP2
```

O custo da junção externa, de acordo com o procedimento acima, seria a soma dos custos dos passos associados (junção interna, projeções e união). Entretanto, observe que o passo 3 pode ser realizado na relação temporária que está sendo construída no passo 2, ou seja, podemos simplesmente completar cada tupla resultante com um *nuil*. Além disso, no passo 4, sabemos que os dois operandos da união são disjuntos (não possuem tuplas em comum), assim, não há necessidade da eliminação de duplicatas.

15.6 COMBINAÇÃO DE OPERAÇÕES USANDO PIPELINES

Em geral, uma consulta em SQL será traduzida em uma expressão da álgebra relacional, que é uma *seqüência de operações relacionais*. Se executarmos uma única operação por vez, deveremos gerar arquivos temporários no disco para manter os resultados dessas operações temporárias, criando uma sobrecarga excessiva. A geração e a ordenação de grandes arquivos temporários consomem muito tempo e podem ser desnecessárias em muitos casos, uma vez que esses arquivos serão imediatamente utilizados como entrada da próxima operação. Para reduzir o número de arquivos temporários, é comum gerar códigos de execução de consulta que correspondam a algoritmos de combinação das operações em uma consulta.

Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

Por exemplo, em vez de ser implementado de maneira separada, um JOIN pode ser obtido pela combinação de duas operações SELECT sobre os arquivos de entrada e uma operação PROJECT final sobre o arquivo resultante; tudo isso é implementado por um algoritmo com dois arquivos de entrada e um único arquivo de saída. Em vez de criar quatro arquivos temporários, aplicamos o algoritmo diretamente e obtemos apenas um arquivo de resultado. Na Seção 15.7.2 veremos como a otimização heurística da álgebra relacional pode agrupar operações para a execução. Essa operação é chamada de *pipelining* ou **processamento baseado em fluxo** (*stream-based*).

É comum criar dinamicamente o código de execução da consulta a fim de implementar múltiplas operações. O código gerado para produzir a consulta combina diversos algoritmos que correspondem às operações individuais. Conforme as tuplas do resultado de uma operação são produzidas, elas são fornecidas como entrada para as operações subseqüentes. Por exemplo, se uma operação de junção for executada após duas operações de seleção sobre relações base, as tuplas resultantes de cada seleção são fornecidas, conforme forem produzidas, como entrada para o algoritmo de junção em um **fluxo** (*stream*) ou *pipeline*.

15.7 UTILIZAÇÃO DE HEURÍSTICAS NA OTIMIZAÇÃO DE CONSULTAS

Nesta seção analisaremos técnicas de otimização que utilizam regras heurísticas para modificar a representação interna de uma consulta — que geralmente está na forma de estrutura de dados de árvore de consulta ou de um grafo de consulta — a fim de melhorar seu desempenho. O analisador sintático de uma consulta de alto nível primeiro gera uma *representação interna inicial*, que depois é otimizada de acordo com regras de heurística. Na seqüência, um plano de execução de consulta é gerado para executar grupos de operações com base nos caminhos de acesso disponíveis para os arquivos envolvidos na consulta.

Uma das principais **regras heurísticas** é aplicar as operações SELECT e PROJECT *antes* de aplicar o JOIN ou outras operações binárias. Isso se deve ao tamanho do arquivo resultante de uma operação binária — tal como o JOIN —, que geralmente é uma função multiplicativa dos tamanhos dos arquivos de entrada. As operações SELECT e PROJECT reduzem o tamanho de um arquivo e, por isso, devem ser aplicadas antes de uma junção ou outra operação binária.

Começaremos na Seção 15.7.1 apresentando as notações de árvore de consulta e de grafo de consulta. Elas podem ser utilizadas como a base para as estruturas de dados que são utilizadas na representação interna de consultas. Uma árvore de consulta é utilizada para representar uma expressão da álgebra relacional ou da álgebra relacional estendida, enquanto um grafo de consulta é usado para representar uma expressão do cálculo relacional. Depois mostraremos na Seção 15.7.2 como as regras de otimização heurísticas são aplicadas para converter uma árvore de consulta em uma **árvore de consulta equivalente**, a qual representa uma expressão diferente da álgebra relacional, que é executada de forma mais eficiente, mas que produz o mesmo resultado que a expressão original. Também analisaremos a equivalência de várias expressões da álgebra relacional. Finalmente a Seção 15.7.3 discutirá a geração de planos de execução de consultas.

15.7.1 Notação de Árvores de Consulta e de Grafos de Consulta

Uma **árvore de consulta** é uma estrutura de dados de árvore que corresponde a uma expressão da álgebra relacional. Ela representa as relações de entrada de uma consulta como *nós folhas* da árvore e representa as operações da álgebra relacional como nós internos. Uma execução de árvore de consulta consiste na execução de uma operação de nó interno sempre que seus operandos estiverem disponíveis — e depois da substituição do nó interno pela relação que resulta da execução da operação. A execução termina quando o nó raiz é executado e produz a relação de resultado da consulta.

A Figura 15.4a mostra uma árvore de consulta para a consulta Q2 dos capítulos 5 a 8: para cada projeto localizado em 'Stafford', recupere o número do projeto, o número do departamento responsável e o último nome, o endereço e a data de nascimento do gerente do departamento. Essa consulta é especificada no esquema relacional da Figura 5.5 e corresponde à seguinte expressão da álgebra relacional:

$\sigma_{\text{PLOCALIZACAO}='STAFFORD'}(\pi_{\text{PNUMERO}, \text{P.DNUM}, \text{E.UNOME}, \text{E.ENDEREÇO}, \text{E.DATANASC}}(\sigma_{\text{DNUM}=\text{DNUMERO}(\text{DEPARTAMENTO})}(\sigma_{\text{GERSSN}=\text{SSN}(\text{EMPREGADO})}))$

Isso corresponde à seguinte consulta SQL:

Q2: SELECT P.PNUMERO, P.DNUM, E.UNOME, E.ENDEREÇO, E.DATANASC FROM PROJETO AS P, DEPARTAMENTO AS D, EMPREGADO AS E WHERE P.DNUM=D.DNUMERO AND D.GERSSN=E.SSN AND P.PLOCALIZACAO='Stafford';

1 5.7 Utilização de Heurísticas na Otimização de Consultas 369

(a)

P.PNUMERO,P.DNUM,E.UNOME,E.ENDERECO,E.DATANASC

(3)

¹D.GERSSN=E.SSN

P.PLOCALIZACAO='Stafford'

(b)

* RPNUMERO,PDNUM,E.UNOME,E.ENDERECO,E.DATANASC

P.DNUM=D.DNUMERO AND D.GERSSN=E.SSN AND RPLOCALIZACAO='Stafford'

(c) [P.PNUMERO.RDNUM]

[E.UNOME,E.ENDERECO,E.DATANASC]

RDNUM=D.DNUMERO ^~\ D.GERSSN=E.SSN

PPLOCALIZACAO='Stafford'

'Stafford'

FIGURA 15.4 Duas árvores de consulta para a consulta Q2. (a) Árvore de consulta correspondente à expressão da álgebra relacional para Q2. (b) Árvore de consulta inicial (canônica) para a consulta SQL Q2. (c) Gráfico de consulta para Q2.

Na Figura 15.4a as três relações PROJETO, DEPARTAMENTO e EMPREGADO são representadas pelos nós folhas P, D e E, enquanto as operações de álgebra relacional da expressão são representadas pelos nós internos da árvore. Quando essa árvore de consulta é executada, o nó rotulado com (1) na Figura 15.4a deve começar a sua execução antes do nó (2), porque algumas tuplas resultantes da operação (1) devem estar disponíveis antes que possamos começar a execução da operação (2). De maneira similar, o nó (2) deve iniciar sua execução e produzir resultados antes que o nó (3) possa começar a sua execução, e assim por diante.

Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

Conforme podemos observar, a árvore de consulta representa uma ordem específica de operações para a execução de uma consulta. Uma representação mais neutra de uma consulta é a notação de grafo de consulta. A Figura 15.4c mostra o grafo de consulta para a consulta Q2. As relações da consulta são representadas pelos nós de relação, que são denotados por círculos de borda única. Valores de constantes, em geral oriundos das condições de seleção da consulta, são representados pelos nós de constante, que são denotados por círculos de borda dupla ou por ovais. As condições de seleção e junção são representadas por arestas do grafo, conforme é mostrado na Figura 15.4c. Finalmente, os atributos a serem recuperados de cada relação são apresentados entre colchetes acima de cada relação.

A representação de grafo de consulta não indica uma ordem na qual as operações devem ser realizadas primeiro. Há apenas um único grafo correspondente a cada consulta. Embora algumas técnicas de otimização sejam baseadas em grafos de consulta, é comumente aceito que as árvores de consulta são preferíveis porque, na prática, o otimizador de consulta precisa mostrar a ordem das operações para a execução da consulta, o que não é possível em grafos de consulta.

15.7.2 Otimização Heurística de Árvores de Consulta

Em geral, muitas expressões diferentes da álgebra relacional — e, por isso, muitas árvores de consulta diferentes — podem ser equivalentes, ou seja, elas podem corresponder à mesma consulta. O analisador sintático irá gerar uma árvore de consulta inicial, que corresponde à consulta SQL, sem fazer nenhuma otimização. Por exemplo, para uma consulta de seleção-projeção-junção, tal como Q2, a árvore inicial é mostrada na Figura 15.4b. Primeiro é aplicado o CARTESIAN PRODUCT (PRODUTO CARTESIANO) das relações especificadas na cláusula FROM, seguido da projeção para os atributos da cláusula SELECT. Tal árvore de consulta canônica representa uma expressão da álgebra relacional que é *muito ineficiente se executada diretamente* por causa das operações CARTESIAN PRODUCT (X). Por exemplo, se as relações PROJETO, DEPARTAMENTO e EMPREGADO tivessem registro de tamanho 100, 50 e 150 bytes, e contivessem 100, 20 e 5.000 tuplas, respectivamente, o resultado do CARTESIAN PRODUCT conteria 10 milhões de tuplas com registros de tamanho de 300 bytes cada. Entretanto, a árvore de consulta da Figura 15.4b é uma forma padrão simples que pode ser criada facilmente. Agora é uma tarefa do otimizador heurístico de consulta transformar essa árvore de consulta inicial em uma árvore de consulta final que seja eficiente para ser executada. O otimizador deve incluir regras de equivalência entre expressões da álgebra relacional que possam ser aplicadas à árvore inicial. Então as regras de heurística para otimização de consultas utilizam essas expressões equivalentes para transformar a árvore inicial na árvore de consulta otimizada final. Primeiro, discutimos informalmente como uma árvore de consulta é transformada por meio do uso de heurísticas. Depois, vimos regras gerais de transformação e mostramos como elas podem ser utilizadas no otimizador algébrico heurístico.

Exemplo de Transformação de uma Consulta. Considere a seguinte consulta Q no banco de dados da Figura 5.5: "Encontre os últimos nomes dos empregados nascidos após 1957 que trabalham no projeto 'Aquarius'". Essa consulta pode ser especificada em SQL como segue:

```
Q: SELECT UNOME
FROM EMPREGADO, TRABALHA_EM, PROJETO WHERE PNOME= 'AQUARIUS' AND PNUMERO=NRP AND ESSN=SSN AND
DATANASC > '31-12-1957';
```

A árvore de consulta inicial para Q é mostrada na Figura 15.5a. Primeiramente a execução direta dessa árvore cria um arquivo muito grande contendo o CARTESIAN PRODUCT dos arquivos inteiros EMPREGADO, TRABALHA_EM e PROJETO. Entretanto, a consulta precisa apenas de um registro da relação PROJETO — o do projeto 'Aquarius' — e dos registros de EMPREGADO daqueles cujas datas de nascimento sejam posteriores a '31-12-1957'. A Figura 15.5b mostra uma árvore de consulta aperfeiçoada que primeiro aplica as operações SELECT para reduzir o número de tuplas que aparecem no CARTESIAN PRODUCT.

Uma melhora mais significativa é alcançada trocando a posição das relações EMPREGADO e PROJETO na árvore, conforme mostrado na Figura 15.5c. Nesse caso, usa-se a informação de que PNUMERO é um atributo-chave da relação projeto e, por isso, a operação SELECT na relação PROJETO irá recuperar apenas um único registro. Podemos melhorar ainda mais a árvore de consulta substituindo qualquer operação CARTESIAN PRODUCT que seja seguida de uma condição de junção por uma operação JOIN, conforme mostrado na Figura 15.5d. Outra melhoria é manter apenas os atributos necessários para as operações subsequentes nas relações intermediárias, por meio da inclusão de operações PROJECT (TT) o mais cedo possível na árvore de consulta, conforme

15 Por isso um grafo de consulta corresponde a uma expressão do *cálculo relacional* (Capítulo 6).

16 Uma consulta também pode ser expressa de várias maneiras em uma linguagem de alto nível tal como a SQL (Capítulo 8).

15.7 Utilização de Heurísticas na Otimização de Consultas **371**

me mostrado na Figura 15.5e. Isso reduz os atributos (colunas) das relações intermediárias, enquanto as operações SELECT reduzem o número de tuplas (registros).

(a)
 UNOME
 PNAME = 'Aquarius' AND PNUMERO = NRP AND ESSN = SSN AND DATANASC > '31-12-1957'
 <^OJETO~^>
 (b)
 UNOME
 PNUMERO = NRP
 PNAME = 'Aquarius'
 DATANASC > '31-12-1957'
 CEMPREGADCT)

FIGURA 15.5 Passos na conversão de uma árvore de consulta durante a otimização heurística. (a) Árvore de consulta inicial (canônica) para a consulta SQL Q. (b) Transferência das operações SELECT para baixo na árvore de consulta.

Como o exemplo anterior demonstra, uma árvore de consulta pode ser transformada passo a passo em outra árvore de consulta mais eficiente de ser executada. Entretanto, devemos nos assegurar de que os passos de transformação sempre levem a uma árvore de consulta equivalente. Para fazê-lo, o otimizador de consulta deve saber quais regras de transformação preservam essa equivalência. Analisaremos algumas dessas regras de transformação a seguir.

372 Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

```
(c)
UNOME
ESSN = SSN
(d)
UNOME
ESSN = SSN
DATANASC > '31-12-1957'
```

FIGURA 15.5 Passos na conversão de uma árvore de consulta durante a otimização heurística, (c) Aplicação, em primeiro lugar, da operação SELECT mais restritiva, (d) Substituindo CARTESIAN PRODUCT e SELECT por operações JOIN. (*continuação*)

Regras Gerais de Transformação para Operações da Álgebra Relacional. Há muitas regras de transformação de expressões da álgebra relacional em operações equivalentes. Aqui estamos interessados no significado dessas operações e nas relações resultantes. Por isso, se duas relações tiverem o mesmo conjunto de atributos em *ordens diferentes*, mas se: duas relações representarem a mesma informação, consideraremos as relações equivalentes. Na Seção 5.1.2 demos uma definição alternativa para *relação* que torna a ordem dos atributos irrelevante; usaremos essa definição aqui. Agora declaramos algumas regras de transformação que são úteis na otimização de consultas, sem prová-las:

1. Cascata de O": Uma condição de seleção conjuntiva pode ser quebrada em uma cascata (ou seja, uma sequência) de operações individuais:

<r

$(R) = a_{c1}(CT_{c2}(\dots((T_{cn}(R))\dots))$

'c1 AND c2 AND ... AND cn'

2. Comutatividade de O": A operação O" é comutativa: $o_{-c1}(cr_{c2}(R)) = o_{-c2}(a_{c1}(R))$

15.7 Utilização de Heurísticas na Otimização de Consultas 373

(e)

UNOME

ESSN = SSN

DATANASC> '31-12-1957'

FIGURA 15.5 Passos na conversão de uma árvore de consulta durante a otimização heurística, (e) Transferência das operações PROJECT para baixo na árvore de consulta, (continuação)

3. Cascata de TT: Em uma cascata (seqüência) de operações TT, todas, exceto a última, podem ser ignoradas:

TT $i^{\wedge}Ustali \cdot \bullet \cdot (TiLisumW)) \cdot \bullet \cdot = \wedge \ll aIW$

ListaiV "Lista2V

4. Comutatividade de O" e TT: Se a condição de seleção c envolver apenas aqueles atributos A_1, \dots, A_n da lista de projeção, as duas operações podem ser comutadas:

TT A_1, A_2, \dots, A_n $\cdot A_n$ $(\langle T_c(R) \rangle) = a_c(TT)$ A_1, A_2, \dots, A_n $\cdot A_n$ (R)

5. Comutatividade de DO (e x): A operação DO é comutativa, assim como a operação x:

 $R \ K_c S = S \ K_c R \ R x S = S x R$

Observe que, embora a ordem dos atributos possa não ser a mesma nas relações resultantes das duas junções (ou dos dois produtos cartesianos), o 'significado' é o mesmo porque a ordem dos atributos não é importante na definição alternativa de relação.

6. Comutatividade de J e DO (ou x): Se todos os atributos da condição de seleção c envolverem apenas os atributos de uma das relações participantes da junção — digamos, R —, as duas operações podem ser comutadas como segue:

 $ff_c(R \ MS) = (CT_c(R)) \ XI \ S$

De maneira alternativa, se a condição de seleção c puder ser escrita como $(c1 \text{ AND } c2)$, na qual a condição $c1$ envolver apenas os atributos de R e a condição $c2$ envolver apenas os atributos de S , as operações comutam da seguinte forma:

 $CT_c(R \ S) = ((T_c(R))N((T_c2(S)))$

As mesmas regras se aplicam se DO for trocado por uma operação x.

7. Comutatividade de TT e DO (OU X): Suponha que a lista de projeção seja $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$, onde A_1, \dots, A_n são atributos de R e B_1, \dots, B_m são atributos de S . Se a condição de junção c envolver apenas atributos que estão em L , as duas operações podem ser comutadas como segue:

 $ir_L(R \ S) - (ir_{A_1, \dots, A_n}(R)) \ *_{\epsilon} (ir_{B_1, \dots, B_m}(S))$

Se a condição de junção c possuir atributos adicionais que não estão em L , estes devem ser acrescidos à lista de projeção, e uma operação TT final é necessária. Por exemplo, se os atributos A_{n+1}, \dots, A_{n+k} de R e B_{m+1}, \dots, B_{m+p} de S

Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

estiverem envolvidos na condição de junção c , mas não estiverem na lista de projeção L , então as operações se comutam da seguinte forma:

$\langle A_n, A_{n+1}, \dots, A_{n+k}$

$(R)) \ N_C(1T_{B_j}, \dots, B_m, B_{m+1}, \dots, B_{m+p}$

(S))) Para x não há condição c , de forma que a primeira regra de transformação sempre se aplica substituindo M_c por x .

8. Comutatividade de operações de conjunto: As operações de conjunto U e CI são comutativas, porém - não é.

9. Associatividade de K , x , U e fl : Essas quatro operações são associativas individualmente, ou seja, se 0 significar qualquer uma dessas quatro operações (na expressão inteira), temos:

$(ReS)0T = Re(S0T)$

10. Comutatividade de $(T$ e das operações de conjunto: A operação cr comuta com as operações U , P_i e $-$. Se 6 significar qualquer uma dessas três operações (na expressão inteira), temos:

$\langle r_c(R \ 9 \ S) \ \langle \langle \langle x_c(R) \rangle \ 9 \ (ff_c(S))$

11. A operação TT comuta com U : $Tr_L(RUS)^*(Tr_L(R))U(TT_L(S))$

12. Conversão de uma sequência (cr, x) em H : Se a condição c de uma cr que siga uma x corresponder a uma condição de junção, converta a sequência (cr, x) em uma X conforme mostrado a seguir:

$CT_c(RXS))EE(RM_cS)$

Há outras transformações possíveis. Por exemplo, uma condição de seleção ou de junção c pode ser convertida em uma condição equivalente por meio do uso das seguintes regras (Leis de DeMorgan):

NOT ($c1$ **AND** $c2$) = (**NOT** $c1$) **OR** (**NOT** $c2$) **NOT** ($c1$ **OR** $c2$) = (**NOT** $c1$) **AND** (**NOT** $c2$)

Transformações adicionais discutidas nos capítulos 5 e 6 não serão repetidas aqui. Veremos a seguir como as transformações podem ser usadas na otimização heurística.

Esboço de um Algoritmo de Otimização Algébrica Heurística. Agora podemos esboçar os passos de um algoritmo que utiliza algumas das regras acima para transformar uma árvore de consulta inicial em uma árvore otimizada que seja mais eficiente para ser executada (na maioria dos casos). O algoritmo irá levar a transformações similares àquelas vistas em nosso exemplo da Figura 15.5. Os passos do algoritmo são os seguintes:

1. A regra 1, ao ser usada, quebra quaisquer operações **SELECT** com condições conjuntivas em uma cascata de operações **SELECT**, permitindo o maior grau de liberdade para transferir operações **SELECT** para ramos diferentes e abaixo na árvore.
2. Usando as regras 2, 4, 6 e 10 relativas à comutatividade do **SELECT** com outras operações, move cada operação **SELECT** o mais longe para baixo na árvore que for permitido pelos atributos envolvidos na condição de seleção.
3. Usando as regras 5 e 9, relativas à comutatividade e associatividade de operações binárias, rearranja os nós folhas da árvore utilizando o seguinte critério: primeiro, posiciona as relações do nó folha com as operações **SELECT** mais restritivas, de forma que elas sejam executadas primeiro na representação de árvore de consulta. A definição de **SELECT mais restritivo** significa aquele que produz uma relação com menor número de tuplas ou com o menor tamanho absoluto. Uma outra possibilidade é definir o **SELECT mais restritivo** como sendo aquele com a menor seletividade; isso é mais prático porque estimativas de seletividade frequentemente estão disponíveis no catálogo do SGBD. Segundo, assegura que a ordem dos nós folhas não cause operações **CARTESIAN PRODUCT**; por exemplo, se duas relações com o **SELECT mais restritivo** não têm condição de junção direta entre elas, pode ser desejável alterar a ordem dos nós folhas para evitar produtos cartesianos.
4. Usando a regra 12, combina uma operação **CARTESIAN PRODUCT** com uma operação **SELECT** subsequente na árvore, gerando uma operação **JOIN** se a condição representa uma condição de junção.
5. Usando as regras 3, 4, 7 e 11, relativas à cascata de **PROJECT** e à comutação de **PROJECT** com outras operações, quebra e transfere as listas de atributos de projeção para baixo na árvore, o mais longe possível, por meio da criação de novas operações **PROJECT** conforme necessário. Apenas aqueles atributos necessários no resultado da consulta e nas operações subsequentes na árvore de consulta devem ser mantidos após cada operação **PROJECT**.
17. Ambas as definições podem ser usadas, uma vez que essas regras são heurísticas.
18. Observe que um produto cartesiano é aceitável em alguns casos — por exemplo, se cada relação possuir apenas uma única tupla porque cada uma tinha, anteriormente, uma condição de seleção em um campo-chave.

15.7 Utilização de Heurísticas na Otimização de Consultas

6. Identifica subárvores que representam grupos de operações que podem ser executadas por um único algoritmo.

Em nosso exemplo, a Figura 15.5b mostra a árvore da Figura 15.5a após a aplicação dos passos 1 e 2 do algoritmo; a Figura 15.5c mostra a árvore após o passo 3; a Figura 15.5d após o passo 4; e a Figura 15.5e após o passo 5. No passo 6 podemos agrupar as operações na subárvore cuja raiz é TT_{ESSN} em um único algoritmo. Também podemos agrupar as operações restantes em uma outra subárvore, na qual as tuplas resultantes do primeiro algoritmo substituem a subárvore cuja raiz é a operação TT_{ESSN} , porque o primeiro agrupamento significa que essa subárvore é executada primeiro.

Sumário de Heurísticas para Otimização Algébrica. Agora resumiremos as heurísticas básicas para a otimização algébrica. A principal heurística é aplicar primeiro as operações que reduzem o tamanho dos resultados intermediários. Isso inclui a realização, o mais cedo possível, de operações **SELECT**, a fim de reduzir o número de tuplas e operações **PROJECT** para diminuir o número de atributos. Isso é feito por meio da transferência de operações **SELECT** e **PROJECT** o mais para baixo possível na árvore. Além disso, as operações **SELECT** e **JOIN**, que são mais restritivas — isto é, resultam em relações com menor número de tuplas ou com menor tamanho absoluto —, devem ser executadas antes de outras operações similares. Isso é feito por meio da reordenação, entre eles próprios, dos nós folhas da árvore, enquanto se evita os produtos cartesianos, e por meio do ajuste apropriado do resto da árvore.

15.7.3 Conversão de Árvores de Consulta em Planos de Execução de Consulta

Um plano de execução para uma expressão de álgebra relacional representada como uma árvore de consulta inclui informações sobre os métodos de acesso disponíveis para cada relação, bem como os algoritmos a serem utilizados na computação dos operadores relacionais representados na árvore. Como um exemplo simples, considere a consulta Q1 do Capítulo 5, cuja expressão correspondente da álgebra relacional é

$^{**}PNOME.UNOME.ENDERE\COV^{**}DNOME.PESQUISA^{**}AS^{**}ATM^{**}) \ ^{**}DNUMERO = DNO EMPREGADO)$

A árvore de consulta é mostrada na Figura 15.6. Para converter isso em um plano de execução, o otimizador poderia escolher um índice de busca para a operação **SELECT** (supondo que exista um), uma varredura de tabela como método de acesso para **EMPREGADO**, um algoritmo de junção de laços aninhados para a junção e uma varredura do resultado do **JOIN** para o operador **PROJECT**. Além disso, a abordagem escolhida para a execução da consulta pode especificar uma avaliação materializada ou por *pipeline*.

"PNAME.UNOME.ENDERECO

^aDNAME='Pesquisa'

EMPREGADO

DEPARTAMENTO **FIGURA 15.6** Uma árvore de consulta para a consulta Q1.

Com a avaliação materializada, o resultado de uma operação é armazenado como uma relação temporária (ou seja, o resultado é *materializado fisicamente*). Por exemplo, a operação de junção pode ser computada e todo o resultado armazenado como uma relação temporária, a qual depois é lida como entrada pelo algoritmo que computa a operação PROJECT, o qual produz a tabela de resultado da consulta. Porém, com a avaliação por *pipeline*, conforme as tuplas resultantes de uma operação são produzidas, elas são encaminhadas diretamente para a próxima operação na sequência da consulta. Por exemplo, conforme as tuplas selecionadas de DEPARTAMENTO são produzidas pela operação SELECT, elas são colocadas em um *buffer*; depois, o algoritmo da operação JOIN consumiria as tuplas do *buffer*, e aquelas tuplas que resultam da operação JOIN são encaminhadas por *pipeline* para o algoritmo da operação de projeção. A vantagem do *pipeline* é a economia de custo, por não ter de escrever os resultados intermediários no disco e por não ter de

lê-los de volta para a próxima operação. 376

Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

15.8 UTILIZAÇÃO DE SELETIVIDADE E ESTIMATIVA DE CUSTO NA OTIMIZAÇÃO DE CONSULTAS

Um otimizador de consultas não deve depender somente de regras heurísticas; ele também deve estimar e comparar os custos da execução de uma consulta usando diferentes estratégias de execução — e deve escolher a estratégia com a *menor estimativa de custo*. Para essa abordagem funcionar, estimativas precisas de custo são necessárias, de forma que diferentes estratégias sejam comparadas de maneira correta e realista. Além disso, devemos limitar o número de estratégias de execução a ser considerado; caso contrário, muito tempo será gasto fazendo as estimativas de custo para as muitas estratégias de execução possíveis. Por isso, essa abordagem é mais adequada para **consultas compiladas**, nas quais a otimização é feita em tempo de compilação e o código da estratégia de execução resultante é armazenado e executado diretamente em tempo de execução. Para **consultas interpretadas**, nas quais todo o processo mostrado na Figura 15.1 ocorre em tempo de execução, uma otimização em escala completa pode diminuir a velocidade do tempo de resposta. Uma otimização mais elaborada é indicada para consultas compiladas, enquanto uma otimização parcial, que consuma menos tempo, funciona melhor para consultas interpretadas.

Chamamos essa abordagem de **otimização de consulta** baseada no **custo**, e ela usa as técnicas tradicionais de otimização que buscam, no *espaço de soluções* para um problema, uma solução que minimize uma função (de custo) objetiva. As funções de custo utilizadas na otimização de consultas são estimativas e não funções exatas de custo, de forma que a otimização pode selecionar uma estratégia de execução de consulta que não seja a estratégia ótima. Na Seção 15.8.1 veremos os componentes do custo da execução de uma consulta. Na Seção 15.8.2 discutiremos o tipo de informação necessária nas funções de custo — informação mantida no catálogo do SGBD. Na Seção 15.8.3 daremos exemplos de funções de custo para a operação SELECT, e na Seção 15.8.4 analisaremos funções de custo para operações JOIN de duas vias. A Seção 15.8.5 discutirá as junções de múltiplas vias e a Seção 15.8.6 dará um exemplo.

15.8.1 Componentes do Custo para a Execução de um Consulta

O custo da execução de uma consulta inclui os seguintes componentes:

1. **Custo de acesso ao armazenamento secundário:** Esse é o custo da busca, da leitura e da escrita de blocos de dados que residem em armazenamento secundário, principalmente em discos. O custo da busca por registros em um arquivo depende dos tipos de estruturas de acesso daquele arquivo, tais como ordenação, *hashing* e índices primários e secundários. Além disso, fatores tais, como se os blocos de arquivo são alocados de maneira adjacente no mesmo cilindro do disco ou se são espalhados no disco, afetam o custo de acesso.
2. **Custo de armazenamento:** Esse é o custo de armazenamento de quaisquer arquivos temporários que sejam gerados por uma estratégia de execução para a consulta.
3. **Custo de computação:** Esse é o custo da realização, na memória, das operações sobre os *buffers* de dados durante a execução da consulta. Tais operações incluem a busca e a ordenação de registros, a fusão (*merging*) de registros em umí junção e a realização de cálculos em valores de campos.
4. **Custo do uso de memória:** Esse é o custo referente ao número de *buffers* de memória necessários durante a execução *da* consulta.
5. **Custo de comunicação:** Esse é o custo do transporte da consulta e de seus resultados de um site de banco de dados para o *site* ou terminal onde a consulta se originou.

Para grandes bancos de dados, a ênfase principal é na minimização do custo de acesso ao armazenamento secundário. Funções simples de custo ignoram outros fatores e comparam as diferentes estratégias de execução da consulta em função de número de transferências de blocos entre o disco e a memória principal. Para bancos de dados menores, nos quais a maioria dos dados dos arquivos envolvidos em uma consulta pode ser armazenada completamente na memória, a ênfase é a minimização do custo de computação. Em bancos de dados distribuídos, nos quais muitos sites estão envolvidos (Capítulo 25), o custo de comunicação também deve ser minimizado. É difícil incluir todos esses componentes de custo em uma função de custo (por derada) por causa da dificuldade em atribuir pesos adequados aos componentes do custo. Essa é a razão pela qual algumas funções de custo consideram apenas um único fator — acesso a disco. Na próxima seção, veremos algumas das informações que são necessárias para a formulação de funções de custo.

19 Essa abordagem foi utilizada pela primeira vez no otimizador do SGBD experimental SYSTEM R, desenvolvido na IBM.

15.8.2 Informações de Catálogo Utilizadas nas Funções de Custo

Para estimar os custos de várias estratégias de execução, devemos manter atualizadas quaisquer informações que sejam necessárias para as funções de custo. Essas informações podem ser armazenadas no catálogo do SGBD, onde são acessadas pelo otimizador de consulta. Primeiro, devemos saber o tamanho de cada arquivo. Para um arquivo cujos registros são todos de um mesmo tipo, também podem ser necessários o **número de registros (tuplas) (r)**, o **tamanho (médio) do registro (R)** e o **número de blocos (b)** (ou boas estimativas deles). O **fator de divisão em blocos (b/r)** para o arquivo também pode ser necessário. Devemos nos manter a par do *método de acesso primário* e dos *atributos de acesso primário* de cada arquivo. Os registros do arquivo podem estar desordenados, ordenados segundo um atributo com ou sem um índice primário ou *dustering*, ou podem estar em um *hash* para um atributo-chave. Também são mantidas informações sobre todos os índices secundários e atributos de indexação. O número de níveis (x) de cada índice multinível (primário, secundário ou *dustering*) é necessário para as funções de custo que estimam o número de acessos a blocos que ocorrem durante a execução da consulta. Em algumas funções de custo, é necessário o **número de blocos de primeiro nível de índice (b_{pi})**.

Um outro parâmetro importante é o **número de valores distintos (d)** de um atributo e sua **seletividade (si)**, que é uma fração dos registros que satisfazem uma condição de igualdade baseada no atributo. Isso permite a estimativa da **cardinalidade** da seleção ($s = si * r$) de um atributo, que é o número médio de registros que irão satisfazer uma condição de seleção de igualdade baseada no atributo. Para um *atributo-chave*, $d = r, si = 1/r$ e $s = 1$. Para um *atributo que não é chave*, por meio da suposição de que os d valores distintos estão distribuídos uniformemente entre os registros, estimamos $si = (1/d)$, portanto, $s = (r/d)$.

Informações como o número de níveis de índice são mantidas facilmente porque elas não se alteram com muita frequência. Entretanto, outras informações podem se alterar frequentemente; por exemplo, o número de registros r de um arquivo se altera toda vez que um registro é incluído ou excluído. O otimizador de consulta precisará de valores razoavelmente próximos a esses parâmetros para uso na estimativa de custo das várias estratégias de execução, mas não necessariamente de valores exatamente atualizados até o último minuto. Nas duas próximas seções, examinamos como alguns desses parâmetros são utilizados em funções de custo de um otimizador de consulta baseado em custos.

15.8.3 Exemplos de Funções de Custo para SELECT

Agora vamos obter as funções de custo para os algoritmos de seleção S1 até S8 discutidos na Seção 15.3.1 em função do *número de transferência de blocos* entre a memória e o disco. Essas funções de custo são estimativas que ignoram o tempo de processamento, o custo de armazenamento e outros fatores. O custo para o método S1 é denotado por C_{S1} acessos a blocos.

- **S1. Abordagem de busca linear (força bruta):** Buscamos em todos os blocos de arquivo para recuperar todos os registros que satisfaçam a condição de seleção, por isso, $C_{S1a} = b$. Para uma condição de igualdade baseada em uma chave, em média, apenas a metade dos blocos de arquivo é pesquisada antes de encontrar o registro, portanto, $C_{S1b} = (b/2)$ se o registro for encontrado; se nenhum registro satisfizer a condição, $C_{S1c} = b$.
- **S2. Busca binária:** Essa busca acessa aproximadamente $C_{S2} = \log_2 b + 1$ (s/bfr) - 1 blocos de arquivo. Isso se reduz a $\log_2 b$ se a condição de igualdade for sobre um atributo único (chave), porque $s = 1$ nesse caso.
- **S3. Uso de um índice primário (S3a) ou uma chave de hash (S3b) para recuperar um único registro:** Para um índice primário, recupera um bloco a mais que o número de níveis de índice, por isso, $C_{S3a} = x + 1$. Para o *hashing*, a função de custo é aproximadamente $C_{S3c} = 1$ para *hashing* estático ou *hashing* linear, e 2 para o *hashing* extensível (Capítulo 13).
- **S4. Uso de um índice ordenado para recuperar múltiplos registros:** Se a condição de comparação for $>, >=, <$ ou $<=$ baseada em um campo-chave com um índice ordenado, de modo grosseiro, a metade dos registros do arquivo irá satisfazer a condição. Isso dá uma função de custo de $C_{S4} = x + (b/2)$. Essa é uma estimativa muito grosseira e, embora possa estar correta na média, ela pode ser bastante imprecisa em casos individuais.
- **S5. Uso de um índice *dustering* para recuperar múltiplos registros:** Dada uma condição de igualdade, s registros irão satisfazer a condição, onde s é a cardinalidade da seleção do atributo de indexação. Isso significa que $\lceil (s/bfr) \rceil$ blocos de arquivo serão acessados, dando $C_{S5} = x + \lceil (s/bfr) \rceil$.
- **S6. Uso de um índice secundário (*árvore-B*):** Em uma comparação de *igualdade*, s registros irão satisfazer a condição, onde s é a cardinalidade da seleção do atributo de indexação. Entretanto, como não é um índice *dustering*, cada um dos registros pode residir em blocos diferentes, de forma que o custo estimado (pior caso) seja $C_{S6a} = x + s$. Isso se reduz a $x + 1$ para um atributo-chave de indexação. Se a condição de comparação for $>, >=, <$ ou $<=$, e se

20 Conforme mencionamos anteriormente, otimizadores mais precisos podem armazenar histogramas da distribuição dos registros segundo os valores de um atributo.

Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

supõe que a metade dos registros do arquivo satisfaça a condição, então (muito grosseiramente) a metade dos blocos de primeiro nível do índice é acessada, mais a metade dos registros do arquivo por meio do índice. A estimativa de custo para esse caso é, aproximadamente, $C_{S6}(c) = x + (\frac{x}{2}) + (r/2)$. A parcela $r/2$ pode ser refinada se melhores estimativas de seletividade estiverem disponíveis.

- S7. *Seleção conjuntiva*: Podemos utilizar tanto SI quanto um dos métodos S2 a S6 discutidos acima. No último caso, usamos uma condição para recuperar os registros e depois verificamos no *buffer* de memória se cada registro recuperado satisfaz as demais condições da conjunção.

- S8. *Seleção conjuntiva utilizando um índice composto*: O mesmo que S3a, S5 ou S6a, dependendo do tipo do índice.

Exemplo do Uso de Funções de Custo. Em um otimizador de consulta, é comum as várias estratégias possíveis para a execução de uma consulta serem enumeradas e os custos das diferentes estratégias serem estimados. Uma técnica de otimização, tal como a programação dinâmica, pode ser utilizada para encontrar eficientemente a estimativa de custo ótima (menor), sem ter de considerar todas as possíveis estratégias de execução. Não analisaremos algoritmos de otimização aqui; em vez disso, usaremos um exemplo simples para ilustrar como as estimativas de custo podem ser utilizadas. Suponha que o arquivo EMPREGADO da Figura 5.5 possua $r_E = 10.000$ registros armazenados em $b_E = 2.000$ blocos de disco com fator de divisão em blocos $bfr_E = 5$ registros/bloco e os seguintes caminhos de acesso:

1. Um índice *clustering* para SALÁRIO, com níveis $x_{SALARIO} = 3$ e cardinalidade média da seleção $s_{SALARIO} = 20$.
2. Um índice secundário para o atributo-chave SSN, com $x_{SSN} = 4$ ($s_{SSN} = 1$).
3. Um índice secundário para o atributo DNO, que não é chave, com $x_{DNO} = 2$ e blocos de índice de primeiro nível $t_{DNO} = 4$. Há $d_{DNO} = 125$ valores distintos para DNO, de forma que a cardinalidade da seleção para DNO é $s_{DNO} = (r_E/d_{DNO}) = 80$.
4. Um índice secundário para SEXO, com $x_{SEXO} = 1$. Há $d_{SEXO} = 2$ valores para o atributo SEXO, de forma que a cardinalidade média da seleção é $s_{SEXO} = (r_E/c_{SEXO}) = 5.000$.

Ilustramos o uso das funções de custo com os seguintes exemplos:

(OP1): CT_{SSN} = 23456789 (EMPREGADO)

(OP2): (X_{DNO} = 5) (EMPREGADO)

(OP3): (X_{DNO} = 5) (EMPREGADO)

^OP4: (X_{DNO} = 5 AND SALARIO > 30.000 AND SEXO = 'F') (EMPREGADO)

O custo da força bruta (busca linear) da opção SI será estimado em $C_{slb} = b_E = 2.000$ (para uma seleção sobre um atributo que não é chave) ou $C_{slb} = (b_E/2) = 1.000$ (custo médio para uma seleção sobre um atributo-chave). Para a OP1 podemos usar tanto o método SI quanto o método S6a; a estimativa de custo para S6a é $C_{S6a} = x_{SSN} + 1 = 4 + 1 = 5$, é escolhida sobre o Método SI, cuja média de custo é $C_{SI} = 1.000$. Para a OP2 podemos usar tanto o método SI (com estimativa de custo $C_{slb} = 2.000$) quanto o método S6b (com estimativa de custo $C_{S6b} = x_{DNO} + (b_{DNO}/2) + (r_E/2) = 2 + (4/2) + (10.000/2) = 5.004$), portanto, escolhemos a abordagem da força bruta para a OP2. Para a OP3 podemos usar tanto o método SI (com estimativa de custo $C_{slb} = 2.000$) quanto o método S6a (com estimativa de custo $C_{S6a} = x_{DNO} + s_{DNO} = 2 + 80 = 82$), portanto, escolhemos o método S6a.

Finalmente, considere a OP4, que possui uma condição de seleção conjuntiva. Precisamos estimar o custo do uso de qualquer um dos três componentes da condição de seleção para recuperar os registros, mais a abordagem da força bruta. A última dá uma estimativa de custo $C_{slb} = 2.000$. Usando primeiramente a condição (DNO = 5), teremos uma estimativa de custo $C_{S6a} = 82$. Usando primeiramente a condição (SALÁRIO > 30.000), uma estimativa de custo $C_{S4} = x_{SALARIO} + (b_E/2) = 3 + (2.000/2) = 1003$. Usando primeiramente a condição (SEXO = 'F'), uma estimativa de custo $C_{S6a} = x_{SEXO} + s_{SEXO} = 1 + 5.000 = 5.001$. Então, o otimizador escolheria o método S6a com o índice secundário sobre DNO porque é o que possui a menor estimativa de custo. A condição (DNO = 5) é usada para recuperar os registros, e a parte restante da condição conjuntiva (SALARIO > 30.000 AND SEXO = 'F') é verificada em cada registro selecionado após ele ser recuperado na memória.

15.8.4 Exemplos de Funções de Custo para JOIN

Para desenvolver funções de custo razoavelmente precisas para operações JOIN, precisamos ter uma estimativa do tamanho (número de tuplas) do arquivo que resulta *após* a operação JOIN. Isso geralmente é mantido como uma proporção do tamanho (número de tuplas) do arquivo de junção resultante em relação ao tamanho do arquivo do produto cartesiano, se ambos são aplicados aos mesmos arquivos de entrada — essa proporção é chamada de seletividade da junção (*js*). Se denotarmos o número de tuplas de uma relação R por I_R , temos

15.8 Utilização de Seletividade e Estimativa de Custo na Otimização de Consultas

379

$$js = \text{IRM}_C SI / \text{IRXS} I = \text{IRM}_C SI / (I RI * SI)$$

Se não houver condição de junção c , então $js = 1$, e a junção é o mesmo que o CARTESIANPRODUCT . Se nenhuma tupla das relações satisfizer a condição de junção, então $js = 0$. Em geral, $0 < js < 1$. Para uma junção na qual a condição c é uma comparação de igualdade $R.A = S.B$, obtemos os dois casos especiais seguintes:

1. Se A for uma chave de R , então $|R \times_c S| < |S|$, portanto $js < (1/|R|)$.
2. Se B for uma chave de S , então $|R \times_c S| < |R|$, portanto $js < (1/|S|)$.

Possuir uma estimativa da seletividade da junção para as condições de junção que ocorrem frequentemente habilita o otimizador de consulta a estimar o tamanho do arquivo resultante após a operação de junção, dados os tamanhos dos dois arquivos de entrada, por meio do uso da fórmula $|R \times_c S| = js * |R| * |S|$. Podemos agora fornecer alguns exemplos de *aproximações* de funções de custo para a estimativa de custo de alguns dos algoritmos de junção dados na Seção 15.3.2. As operações de junção são da forma $R \bowtie_{A=B} S$

onde A e B são atributos compatíveis com o domínio de R e S , respectivamente. Suponha que R possua b_R blocos e que S possua b_S blocos:

- J1. *Junção de laços aninhados*: Suponha que usamos R para o laço externo; então obteremos a seguinte função de custo para estimar o número de acessos a blocos para esse método, supondo três *buffers de memória*. Suponhamos que o fator de divisão em blocos para o arquivo resultante é bfr_{RS} , e que a seletividade da junção seja conhecida: $C_n = b_R + (b_R * b_S) + ((js * |R| * |S|) / bfr_{RS})$

A última parte da fórmula é o custo de gravação do arquivo resultante no disco. Essa fórmula de custo pode ser modificada para levar em consideração diferentes números de *buffers* de memória, conforme foi discutido na Seção 15.3.2.

- J2. *Junção de laço único [uso de uma estrutura de acesso para recuperar o(s) registro(s) correspondente(s)]*: Se um índice existir para o atributo de junção B de S com x_B níveis de índice, poderemos recuperar cada registro s de R e depois usar o índice para recuperar todos os correspondentes registros t de S que satisfaçam $t[B] = s[A]$. O custo depende do tipo do índice. Para um índice secundário, no qual s_B é a cardinalidade da seleção para o atributo de junção B de S , obtemos

$$C_{J2a} = b_R + (|R| * (x_B + s_B)) + ((js * |R| * |S|) / bfr_{RS})$$

Para um índice *clustering* no qual s_B é a cardinalidade da seleção de B , obtemos

$$C_{J2b} = h + (|R| * (x_B + (s_B / bfr_B))) + ((js * |R| * |S|) / bfr_{RS})$$

Para um índice primário, obtemos

$$C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|) / bfr_{RS})$$

Se uma chave de *hash* existir para um dos dois atributos da junção — digamos, B de S — obtemos

$$C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|) / bfr_{RS})$$

onde $h > 1$ é o número médio de acessos a blocos para recuperar um registro, dado seu valor de chave de *hash*.

- J3. *Junção sort-merge*: Se os arquivos já estiverem ordenados segundo os atributos da junção, a função de custo para esse método será

$$C_{J3a} = f_R + f_S + (|R| * |S| / bfr_{RS})$$

Se devemos ordenar os arquivos, o custo da ordenação deve ser adicionado. Podemos usar as fórmulas da Seção 15.2 para estimar o custo da ordenação.

21 A *cardinalidade da seleção* foi definida como sendo o número médio de registros que satisfazem uma condição de igualdade baseada no atributo, que é o número médio de registros que possuem o mesmo valor para o atributo e , por isso, irão participar da junção com um único registro do outro arquivo.

380 Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

Exemplo de Uso das Funções de Custo. Suponha que temos o arquivo EMPREGADO descrito no exemplo da seq anterior, e considere que o arquivo DEPARTAMENTO da Figura 5.5 consista de $r_D = 125$ registros armazenados em $b_D = 13$ blocos i disco. Considere as operações de junção:

(OP6): EMPREGADO $\bowtie_{\text{DNO}=\text{DNUMERO}}$ DEPARTAMENTO (OP7): DEPARTAMENTO $\bowtie_{\text{NGERSSN}=\text{SSN}}$ EMPREGADO

Suponha que temos um índice primário para DNUMERO de DEPARTAMENTO com $x_{\text{DNUMERO}} = 1$ nível e um índice secundário pa GERSSN de DEPARTAMENTO com cardinalidade de seleção $s_{\text{GERSSN}} = 1$ e $\%_{\text{BSSN}} = 2$ níveis. Suponha que a seletividade da junção para O] seja $j_{\text{OP6}} = (1 / I_{\text{DEPARTAMENTO}} I) = 1/125$, porque DNUMERO é uma chave de DEPARTAMENTO. Suponha também que o fator de divisi em blocos para o arquivo de junção resultante seja $b_{fr_{ED}} = 4$ registros por bloco. Podemos estimar os custos do pior caso parc operação JOIN OP6 usando os métodos aplicáveis J1 e J2 conforme segue:

1. Usando o Método J1 com EMPREGADO como laço externo:

$$C_{J1} = b_E + (b_E * b_D) + ((j_{\text{OP6}} * r_E * r_D) / b_{fr_{ED}}) \\ = 2.000 + (2.000 * 13) + (((1/125) * 10.000 * 125) / 4) = 30.500$$

2. Usando o Método J1 com DEPARTAMENTO como laço externo:

$$C_{J1} = b_D + (b_E * b_D) + (0's_{\text{OP6}} * 'E * r_D) / b_{fr_{ED}} \\ = 13 + (13 * 2.000) + (((1/125) * 10.000 * 125) / 4) = 28.513$$

3. Usando o Método J2 com EMPREGADO como laço externo: $C_{J2} = b_E + (r_E * (x_{\text{DNUMERO}} + 1)) + (0's_{\text{OP6}} * 'E * r_D) / b_{fr_{ED}}$

$$= 2.000 + (10.000 * 2) + (((1/125) * 10.000 * 125) / 4) = 24.500$$

4. Usando o Método J2 com DEPARTAMENTO como laço externo:

$$C_{J2} = b_D + (r_D * (x_{\text{DNO}} + s_{\text{DNO}})) + (0's_{\text{OP6}} * 'E * r_D) / b_{fr_{ED}} \\ = 13 + (125 * (2 + 80)) + (((1/125) * 10.000 * 125) / 4) = 12.763$$

O caso 4 possui a menor estimativa de custo e será escolhido. Observe que se 15 (ou mais) buffers de memória estive sem disponíveis para a execução da junção em vez de apenas três, 13 deles poderiam ser usados para manter a relação DEPARTAMENTO inteiramente na memória, um poderia ser usado como buffer para o resultado, e o custo do caso 2 poderia ser drasticamente reduzido para apenas $b_E + b_D + ((j_{\text{OP6}} * r_E * r_D) / b_{fr_{ED}})$ ou 4.513, conforme visto na Seção 15.3.2. Como exercício o leitor poderá realizar uma análise similar para a OP7.

15.8.5 Consultas de Múltiplas Relações e Ordem de Junções

As regras de transformação algébrica da Seção 15.7.2 incluem uma regra comutativa e uma regra associativa para a operação de junção. Com essas regras, muitas expressões de junção equivalentes podem ser produzidas. Como resultado, o número de árvores de consulta alternativas cresce muito rapidamente conforme cresce o número de junções em uma consulta. Em geral, uma consulta que realiza a junção de n relações terá $n - 1$ operações de junção e, por isso, pode ter um grande número de: dens de junção diferentes. Estimar o custo de cada árvore de junção possível para uma consulta com um grande número de junções irá exigir uma quantidade de tempo substancial pelo otimizador de consulta. Por isso, é necessário descartar possíveis árvores de consulta. Em geral, os otimizadores de consulta limitam a estrutura de uma árvore de consulta (junção) àquelas árvores profundas à esquerda (ou profundas à direita). Uma árvore profunda à esquerda é uma árvore binária, cujo filho à direita para cada nó que não é folha é sempre uma relação base. O otimizador poderia escolher a árvore profunda à esquerda, em particular, com o menor custo estimado. Dois exemplos de árvores profundas à esquerda são mostrados na Figura 15.1 (Observe que as árvores da Figura 15.5 também são árvores profundas à esquerda.)

Com árvores profundas à esquerda, o filho à direita é considerado ser a relação interna quando da execução da junção de laços aninhados. Uma vantagem das árvores profundas à esquerda (ou profundas à direita) é que elas são receptivas para *pipelining*, conforme discutido na Seção 15.6. Por exemplo, considere a primeira árvore profunda à esquerda da Figura 15.7 suponha que o algoritmo de junção é o método de laço único; nesse caso, uma página de disco de tuplas da relação externa utilizada para pesquisar as tuplas correspondentes na relação interna. Como um bloco de tuplas resultantes é produzido a partir da junção de R_1 e R_2 , ele poderia ser usado para pesquisar em R_3 . Da mesma forma, como uma página de tuplas resultante

15.8 Utilização de Seletividade e Estimativa de Custo na Otimização de Consultas 381

é produzida a partir dessa junção, ela poderia ser usada para pesquisar em *R4*. Uma outra vantagem das árvores profundas à esquerda (ou profundas à direita) é que possuem uma relação base como uma das entradas de cada junção possibilita ao otimizador utilizar para aquela relação quaisquer caminhos de acesso que possam ser úteis na execução da junção.

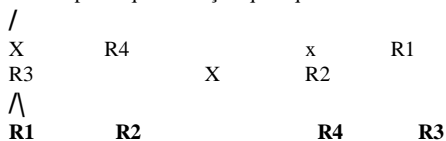


FIGURA 15.7 Duas árvores de consulta (junção) profundas à esquerda.

Se for utilizada a materialização em vez de *pipelining* (Seção 15.6), os resultados da junção poderiam ser materializados e armazenados como relações temporárias. A idéia-chave sob o ponto de vista do otimizador em relação à ordem das junções é encontrar uma ordem que irá reduzir o tamanho dos resultados intermediários, uma vez que os resultados intermediários (por *pipe-line* ou materializados) são utilizados pelos operadores subsequentes e, por isso, afetam o custo da execução daqueles operadores.

15.8.6 Exemplo para Ilustrar a Otimização de Consulta Baseada em Custo

Consideraremos a consulta Q2 e sua árvore de consulta mostrada na Figura 15.4a para ilustrar a otimização de consulta baseada em custo:

Q2: SELECT PNUMERO, DNUM, UNOME, ENDEREÇO, DATANASC FROM PROJETO, DEPARTAMENTO, EMPREGADO WHERE DNUM=DNUMERO AND GERSSN=SSN AND PLOCALIZACAO='Stafford';

Suponha que temos a informação estatística sobre as relações mostradas na Figura 15.8. As estatísticas MENOR_VALOR e MAIOFLVALOR foram normalizadas por questões de clareza. Supõe-se que a árvore da Figura 15.4a represente o processo de otimização algébrica heurística e o início da otimização baseada em custo (neste exemplo, supomos que o otimizador heurístico não transfere as operações de projeção para baixo na árvore).

Em primeiro lugar, a otimização baseada em custo considera a ordem das junções. Conforme mencionado anteriormente, supomos que o otimizador considera apenas árvores profundas à esquerda, portanto, as ordens potenciais para as junções — sem produto cartesiano — são:

1. PROJETO DO DEPARTAMENTO DO EMPREGADO
2. DEPARTAMENTO DO PROJETO DO EMPREGADO
3. DEPARTAMENTO DO EMPREGADO DO PROJETO
4. EMPREGADO DO DEPARTAMENTO DO PROJETO

Suponha que a operação de seleção já tenha sido aplicada à relação PROJETO. Se supusermos uma abordagem de materialização, então uma nova relação temporária será criada após cada operação de junção. Para examinar o custo da ordem de junções (1), a primeira junção é entre PROJETO e DEPARTAMENTO. Os métodos de junção e de acesso para as relações de entrada devem ser determinados. Uma vez que de acordo com a Figura 15.8 DEPARTAMENTO não possui índice, o único método de acesso disponível é a varredura da tabela (ou seja, a busca linear). A relação PROJETO terá a operação de seleção executada antes da junção, portanto, há duas opções: varredura da tabela (busca linear) ou a utilização de seu índice PROJ_PLOC, portanto, o otimizador deve comparar seus custos estimados. A informação estatística para o índice PROJ_PLOC (Figura 15.8) mostra que o número de níveis do índice é $x = 2$ (raiz mais níveis folha). O índice não é único (porque PLOCALIZACAO não é uma chave de PROJETO), portanto, o otimizador supõe uma distribuição uniforme de dados e estima que o número de ponteiros de registros para cada valor de PLOCALIZACAO seja 10. Isso é calculado a partir das tabelas da Figura 15.8 por meio da multiplicação SELETIVIDADE * NUMJNHAS, onde SELETIVIDADE é estimada por $1/\text{NUM_DISTINTOS}$. Portanto, o custo do uso do índice e do acesso aos registros é estimado em 12 acessos a blocos (2 para o índice e 10 para os blocos de dados). O custo de uma varredura da tabela é estimado em 100 acessos a blocos, portanto, o acesso pelo índice é mais eficiente conforme se esperava.

382 Capítulo 15 Algoritmos para Processamento e Otimização de Consultas (a) NOME_DA_TABELA

NOME_DA_COLUNA	NUM_DISTINTOS	MENOR_VALOR	MAIOR_VALOR
PROJETO	PLOCALIZACAO	200	200
PROJETO	PNUMERO	2.000	2.000
PROJETO	DNUM	50	50
DEPARTAMENTO	DNUMERO	50	50
DEPARTAMENTO	GERSSN	50	50
EMPREGADO	SSN	10.000	10.000
EMPREGADO	DNO	50	50
EMPREGADO	SALÁRIO	500	500
(b) NOME_DA_TABELA	NUMJNHAS	BLOCOS	
PROJETO	2.000	100	
DEPARTAMENTO	50	5	
EMPREGADO	10.000	2.000	
(c) NOMEJX3NDICE	UNICIDADE	NIVELB*	BLOCOS_FOLHAS
CHAVES_DISTINTAS			
PROJ_PLOC	NAO_UNICA	1	4
EMP_SSN	UNICA	1	50
EMP_SAL	NAOJNICA	1	50

*NIVELB é o número de níveis sem o nível folha.

FIGURA 15.8 Amostra de informações estatísticas para as relações deQ2. (a) Informações de colunas, (b) Informações de tab las. (c) Informações de índice.

Na abordagem de materialização, um arquivo temporário TEMPI do tamanho de 1 bloco é criado para manter o resultac da operação de seleção. O tamanho do arquivo é calculado pela determinação do fator de divisão em blocos usando a fórmula $\text{NUM_LINHAS}/\text{BLOCOS}$, que dá $2.000/100$, ou 20 linhas por bloco. Por isso, os 10 registros selecionados da relação PROJETO cab rão em um único bloco. Agora podemos calcular o custo estimado da primeira junção. Consideraremos apenas o método < junção de laços aninhados, no qual a relação externa é o arquivo temporário, TEMPI, e a relação interna é DEPARTAMENTO. Uma v que o arquivo TEMPI inteiro cabe no espaço de *buffer* disponível, precisamos ler cada um dos cinco blocos da tabela DEPARTAMEN apenas uma vez, portanto, o custo da junção é seis acessos a blocos mais o custo da escrita do arquivo temporário de resultad TEMP2. O otimizador teria de determinar o tamanho de TEMP2. Uma vez que o atributo de junção DNUMERO é a chave < DEPARTAMENTO, qualquer valor de DNUM de TEMPI irá participar da junção com no máximo um registro de DEPARTAMENTO, portanto, número de linhas em TEMP2 será igual ao número de linhas em TEMPI, que é 10. O otimizador determinaria o tamanho do regi tro de TEMP2 e o número de blocos necessários para armazenar essas 10 linhas. Por concisão, suponha que o fator de divisão e blocos para TEMP2 seja de cinco linhas por bloco, portanto, um total de dois blocos é necessário para armazenar TEMP2. Finalmente, o custo da última junção precisa ser estimado. Podemos usar uma junção de laço único em TEMP2, uma v que nesse caso o índice EMP_SSN (Figura 15.8) pode ser usado para pesquisar e localizar os registros correspondentes e: EMPREGADO. Por isso, o método de junção envolveria a leitura de cada bloco de TEMP2 e a procura por cada um dos cinco valores c GERSSN usando o índice EMP_SSN. Cada pesquisa em índice exigiria um acesso à raiz, um acesso à folha e um acesso ao bloco c dados ($x + 1$, onde o número de níveis x é 2). Portanto, 10 pesquisas exigem 30 acessos a blocos. Acrescentando os dois ace sos a blocos para TEMP2, teremos um total de 32 acessos a blocos para essa junção.

Para a projeção final, suponha que o *pipelining* é utilizado para produzir o resultado final, o que não exige acessos adicic nais a blocos, portanto, o custo total da ordem de junções (1) é estimado como o somatório dos custos anteriores. Depois otimizador estimaria de maneira similar os custos das outras três ordens de junções e escolheria aquela com a menor estimat va. Deixamos isso como um exercício para o leitor.

15.9 VISÃO GERAL DA OTIMIZAÇÃO DE CONSULTAS NO ORACLE

O SGBD ORACLE (Versão 7) fornece duas abordagens diferentes para a otimização de consultas: baseada em regras e cust(Com a abordagem baseada em regras, o otimizador escolhe os planos de execução baseando-se em operações classificadas hí

urísticamente. A ORACLE mantém uma tabela de 15 caminhos de acesso classificados, na qual uma posição mais baixa na classificação implica uma abordagem mais eficiente. Os caminhos de acesso variam do acesso à tabela pelo ROWID (mais eficiente) — em que ROWID (identificador de linha) especifica o endereço físico do registro incluindo arquivo de dados, bloco de dados e deslocamento de linha dentro do bloco — a uma varredura completa da tabela (menos eficiente) — na qual todas as linhas são pesquisadas por meio de leituras de blocos múltiplos. Entretanto, a abordagem baseada em regras está sendo substituída gradualmente pela abordagem baseada em custo, na qual o otimizador examina caminhos de acesso alternativos e algoritmos de operadores, e escolhe o plano de execução com menor estimativa de custo. O custo estimado da consulta é proporcional ao tempo decorrido esperado para executar a consulta segundo um dado plano de execução. O otimizador da ORACLE calcula este custo com base no uso estimado dos recursos, tais como a necessidade de ENTRADA/SAÍDA, de tempo de CPU e de memória. O objetivo da otimização baseada em custo na ORACLE é minimizar o tempo decorrido para processar a consulta inteira.

Uma funcionalidade interessante do otimizador de consultas da ORACLE é a capacidade de um desenvolvedor de aplicações definir dicas para o otimizador. A idéia é que um desenvolvedor de aplicações saberia mais informações sobre os dados do que o otimizador. Por exemplo, considere a tabela EMPREGADO mostrada na Figura 5.5. A coluna SEXO da tabela possui apenas dois valores distintos. Se existirem 10.000 empregados, então o otimizador estimaria que metade é masculina e metade é feminina, supondo uma distribuição uniforme dos dados. Se existir um índice secundário, ele muito provavelmente não seria utilizado. Entretanto, se o desenvolvedor da aplicação souber que existem apenas 100 empregados masculinos, uma dica poderia especificar que, em uma consulta SQL, cuja condição da cláusula WHERE é SEXO = 'M', o índice associado fosse utilizado no processamento da consulta. Várias dicas podem ser especificadas, tais como:

- A abordagem de otimização para uma sentença SQL.
- O caminho de acesso para uma tabela acessada por uma sentença.
- A ordem de junção para uma sentença de junção.
- Uma operação de junção em particular em uma sentença de junção.

A otimização baseada em custo na Oracle 8 é um bom exemplo de uma abordagem sofisticada aplicada para otimizar consultas SQL em SGBDRs comerciais.

15.10 OTIMIZAÇÃO SEMÂNTICA DE CONSULTAS

Uma abordagem diferente para a otimização de consultas, chamada otimização semântica de consultas, foi sugerida. Essa técnica, que pode ser usada em combinação com as técnicas discutidas anteriormente, usa restrições especificadas no esquema do banco de dados — tais como atributos únicos e outras restrições mais complexas —, a fim de transformar uma consulta em uma outra consulta que seja mais eficiente para ser executada. Não analisaremos essa abordagem em detalhes, apenas a ilustraremos com um exemplo simples. Considere a consulta SQL:

```
SELECT E.UNOME, G.UNOME
FROM EMPREGADO AS E, EMPREGADO AS G
WHERE E.SSN_SUPER=G.SSN AND E.SALARIO>G.SALARIO
```

Essa consulta recupera os nomes dos empregados que ganham mais que seus supervisores. Suponha que tivéssemos uma restrição no esquema do banco de dados que declara que nenhum empregado pode ganhar mais que seu supervisor direto. Se o otimizador semântico de consultas verificar a existência dessa restrição, ele absolutamente não precisaria executar a consulta, porque ele sabe que o resultado da consulta será vazio. Isso pode economizar um tempo considerável se a verificação da restrição puder ser feita eficientemente. Entretanto, pesquisar pelas muitas restrições para encontrar aquelas que são aplicáveis a uma determinada consulta e que podem otimizá-la semanticamente também pode consumir bastante tempo. Com a inclusão de regras ativas nos sistemas de bancos de dados (Capítulo 24), as técnicas de otimização semântica de consultas podem eventualmente ser completamente incorporadas aos SGBDs do futuro.

15.11 RESUMO

Neste capítulo demos uma visão geral das técnicas utilizadas por SGBDs no processamento e na otimização de consulta de alto nível. Primeiro vimos como consultas SQL são traduzidas para a álgebra relacional; depois, como várias operações da álgebra

22 Tais dicas também foram chamadas de *anotações* da consulta.

384

Capítulo 15 Algoritmos para Processamento e Otimização de Consultas

relacional podem ser executadas por um SGBD. Vimos que algumas operações, particularmente SELECT e JOIN, podem ter muitas opções de execução. Também vimos como as operações podem ser combinadas durante o processamento da consulta para criar uma execução baseada em *pipeline* ou fluxo (*stream*) em vez de na execução materializada.

Na sequência, descrevemos abordagens heurísticas para a otimização de consultas, as quais usam regras heurísticas e técnicas algébricas para melhorar a eficiência da execução da consulta. Mostramos como uma árvore de consulta, que representa uma expressão da álgebra relacional, pode ser otimizada heurísticamente por meio da reorganização dos nós da árvore da sua transformação em uma outra árvore de consulta equivalente, que é mais eficiente para ser executada. Também demos regras de transformação que preservam a equivalência e que podem ser aplicadas a uma árvore de consulta. Depois, apresentamos planos de execução de consultas para consultas SQL, os quais acrescentam métodos de planos de execução às operações das árvores de consulta.

Analisamos também a abordagem para a otimização de consultas baseada em custo. Mostramos como as funções de custo são desenvolvidas para alguns algoritmos de acesso a bancos de dados e como essas funções de custo são utilizadas para estimar os custos de diferentes estratégias de execução. Apresentamos uma visão geral do otimizador de consultas da ORACLE e mencionamos a técnica de otimização semântica de consultas.

Questões para Revisão

15.1 Discuta as razões para a conversão de consultas SQL em consultas da álgebra relacional antes que a otimização seja realizada. Discuta os diferentes algoritmos para a implementação de cada um dos seguintes operadores relacionais e as circunstâncias sob as quais cada algoritmo pode ser utilizado: SELECT, JOIN, PROJECT, UNION, INTERSECTION, SET DIFFERENCE e CARTESIAN PRODUCT. O que é plano de execução de consulta?

Qual o significado do termo *otimização heurística*? Discuta as principais heurísticas que são aplicadas durante a otimização de consultas.

Como uma árvore de consulta representa uma expressão da álgebra relacional? Qual o significado de uma execução de uma árvore de consulta? Discuta as regras de transformação de árvores de consulta e identifique quando cada regra poderia ser aplicada durante a otimização.

Quantas ordens de junções diferentes existem para uma consulta que faça a junção de dez relações? Qual o significado de *otimização de consultas baseada em custo*? Qual a diferença entre *pipelining* e *materialização*?

Análise os componentes de custo para uma função de custo que é utilizada para estimar o custo da execução de consulta. Quais componentes de custo são usados mais frequentemente como base para as funções de custo?

15.10 Discuta os diferentes tipos de parâmetros que são usados nas funções de custo. Onde essas informações são mantidas?

15.11 Liste as funções de custo para os métodos SELECT e JOIN apresentados na Seção 15.8.

15.12 Qual o significado de otimização semântica de consultas? Como ela difere das outras técnicas de otimização de consultas?

15.2

15.3 15.4

15.5

15.6 15.7 15.8 15.9

Exercícios

15.13 Considere as consultas SQL Q1, Q8, Q1B, Q4 e Q27 do Capítulo 8.

a. Desenhe pelo menos duas árvores de consulta que podem representar cada uma dessas consultas. Sob quais circunstâncias você utilizaria cada uma de suas árvores de consulta?

b. Desenhe a árvore de consulta inicial para cada uma dessas consultas, depois mostre como a árvore de consulta é otimizada pelo algoritmo esboçado na Seção 15.7.

c. Para cada consulta, compare suas árvores de consulta do item (a) com as árvores de consulta inicial e final do item (b).

15.14 Um arquivo com 4.096 blocos será ordenado em um espaço de *buffer* disponível de 64 blocos. Quantas passagens serão necessárias na fase de fusão (*merge*) do algoritmo *sort-merge* (ordenação-fusão) externo?

15.15 Desenvolva funções de custo para os algoritmos PROJECT, UNION, INTERSECTION, SET DIFFERENCE e CARTESIAN PRODUCT vistos na Seção 15.4.

15.16 Desenvolva funções de custo para um algoritmo que consista de dois SELECTs, um JOIN e um PROJECT final, em função das funções de custo das operações individuais.

15.17 Pode um índice esparsos ser utilizado na implementação de um operador de agregação? Responda porque sim ou porque não.

15.11 Resumo 385

15.18 Calcule as funções de custo para diferentes opções de execução da operação JOIN OP7 discutidas na Seção 15.3.2.

15.19 Desenvolva fórmulas para o algoritmo de junção *hash* híbrido para o cálculo do tamanho do *buffer* para o primeiro *bucket*. Desenvolva fórmulas mais precisas de estimativa do custo para o algoritmo.

15.20 Estime o custo das operações OP6 e OP7, utilizando as fórmulas desenvolvidas no Exercício 15.19.

15.21 Estenda o algoritmo de junção *sort-merge* (ordenação-fusão) para implementar a operação de junção externa à esquerda.

15.22 Compare o custo de dois planos de consulta diferentes para a seguinte consulta:

⁰⁰SALARIO > 40.000 (EMPREGADO MENDI-DNUMERO DEPARTAMENTO)

Use as estatísticas de bancos de dados da Figura 15.8.

Bibliografia Seleccionada

Um levantamento (*survey*) realizado por Graefe (1993) discute a execução de consultas em sistemas de bancos de dados e inclui uma extensa bibliografia. Um levantamento (*survey*) em formato de artigo realizado por Jarke e Koch (1984) fornece uma taxonomia da otimização de consultas e inclui uma bibliografia de trabalhos nessa área. Um algoritmo detalhado para a otimização da álgebra relacional é dado em Smith e Chang (1975). A tese de doutorado de Kooi (1980) fornece a base para as técnicas de processamento de consultas.

Whang (1985) discute a otimização de consultas em OBE (*Office-By-Example*), que é um sistema baseado em QBE. A otimização baseada em custos foi introduzida no SGBD experimental SYSTEM R e é vista em Astrahan *et al.* (1976). Selinger *et al.* (1979) analisam a otimização de junções de múltiplas vias no SYSTEM R. Algoritmos de junção são discutidos em Gotlieb (1975), Blasgen e Eswaran (1976) e Whang *et al.* (1982). Os algoritmos de *hash* para a implementação de junções são descritos e analisados em DeWitt *et al.* (1984), Bratbergsengen (1984), Shapiro (1986), Kitsuregawa *et al.* (1989), e Blakeley e Martin (1990), entre outros. As abordagens para encontrar uma boa ordem de junção são apresentadas em Ioannidis e Kang (1990) e em Swami e Gupta (1989). Uma discussão sobre as implicações das árvores de junção profundas à esquerda e árvores de junção espessas (*bushy*) é apresentada em Ioannidis e Kang (1991). Kim (1992) discute as transformações de consultas SQL aninhadas em representações canônicas. A otimização de funções de agregação é analisada em Klug (1982) e Muralikrishna (1992). Salzberg *et al.* (1990) descrevem um algoritmo rápido de ordenação externa. A estimativa do tamanho das relações temporárias é crucial para a otimização de consultas. Esquemas de estimativa baseados em amostra são apresentados em Hass *et al.* (1995) e em Haase Swami (1995). Liptonet *et al.* (1990) também discutem a estimativa de seletividade. Fazer o sistema de banco de dados armazenar e usar estatísticas mais detalhadas na forma de histogramas é o tópico de Muralikrishna e DeWitt (1988) e Poosala *et al.* (1996).

Kim *et al.* (1985) discutem tópicos avançados da otimização de consultas. A otimização semântica de consultas é discutida em King (1981) e Malley e Zdonick (1986). Trabalho mais recente sobre a otimização semântica de consultas é relatado em Chakravarthy *et al.* (1990), Shenoy e Ozsoyoglu (1989), e Siegel *et al.* (1992).

!••£ 16

Projeto e Sintonização (*Tuning*) de Bancos de Dados na Prática

Neste capítulo primeiro veremos, na Seção 16.1, questões que surgem no projeto físico de bancos de dados. Depois discutiremos, na Seção 16.2, como melhorar o desempenho por meio da sintonização (*tuning*) do banco de dados.

16.1 PROJETO FÍSICO DE BANCOS DE DADOS EM BANCOS DE DADOS RELACIONAIS

Nesta seção primeiro discutiremos os fatores do projeto físico que afetam o desempenho de aplicações e transações; a seguir comentaremos orientações específicas para SGBDRs.

16.1.1 Fatores que Influenciam o Projeto Físico de Bancos de Dados

O projeto físico é uma atividade na qual o objetivo não é apenas obter uma estrutura de dados apropriada para armazenamento, mas desenvolvê-lo de maneira que garanta um bom desempenho. Para um dado esquema conceitual, há muitas alternativas de projeto físico em um determinado SGBD. Não é possível tomar decisões de projeto físico e realizar análises de desempenho significativas até que conheçamos as consultas, as transações e as aplicações que são esperadas para ser executadas no banco de dados. Devemos analisar essas aplicações, suas frequências esperadas de chamadas, quaisquer restrições de tempo para suas execuções e a frequência esperada de operações de atualização. Veremos cada um desses fatores a seguir.

A. Análise das Consultas e Transações no Banco de Dados. Antes de empreender o projeto físico do banco de dados, devemos ter uma boa idéia do uso pretendido para o banco de dados por meio da definição das consultas e das transações que esperamos ser executadas no banco de dados em formato de alto nível. Para cada consulta devemos especificar o seguinte:

1. Os arquivos que serão acessados pela consulta.
2. Os atributos sobre os quais quaisquer condições de seleção para a consulta são especificadas.
3. Os atributos sobre os quais quaisquer condições de junção ou condições para ligar múltiplas tabelas ou objetos para a consulta são especificadas.
- 4- Os atributos cujos valores serão recuperados pela consulta.

1 Para simplificar, usamos o termo *arquivos*, que pode ser substituído por tabelas, classes ou objetos.

1 6.1 Projeto Físico de Bancos de Dados em Bancos de Dados Relacionais

387

Os atributos listados anteriormente nos itens 2 e 3 são candidatos para a definição de estruturas de acesso. Para cada transação ou operação de atualização devemos especificar o seguinte:

1. Os arquivos que serão atualizados.
2. O tipo de operação em cada arquivo (inclusão, atualização ou exclusão).
3. Os atributos sobre os quais as condições de seleção para uma exclusão ou para uma atualização são especificadas.
4. Os atributos cujos valores serão alterados por uma operação de atualização.

Novamente, os atributos listados anteriormente no item 3 são candidatos para estruturas de acesso. Porém, os atributos listados no item 4 são candidatos a ser evitados em uma estrutura de acesso, uma vez que sua modificação exigirá a atualização das estruturas de acesso.

B. Análise da Frequência Esperada das Chamadas de Consultas e Transações. Além da identificação das características das consultas e das transações esperadas, devemos considerar suas taxas esperadas de chamadas. Essa informação de frequência, aliada às informações de atributos coletadas para cada consulta e transação, é utilizada para compilar uma lista acumulativa da frequência esperada de uso para todas as consultas e transações. Isso é expresso como a frequência esperada de uso de cada atributo em cada arquivo como um atributo de seleção ou um atributo de junção, para todas as consultas e transações. Geralmente, para grandes volumes de processamento, se aplica a 'regra' informal '80-20', a qual estabelece que aproximadamente 80% está associado a apenas 20% das consultas e das transações. Portanto, em situações práticas, raramente é necessário coletar estatísticas e taxas de chamadas exaustivamente para todas as consultas e transações; é suficiente determinar os 20%, ou seja, os mais importantes.

C. Análise das Restrições de Tempo das Consultas e das Transações. Algumas consultas e transações podem possuir restrições rigorosas de desempenho. Por exemplo, uma transação pode ter a restrição de que deva terminar dentro de 5 segundos em 95% das ocasiões em que é chamada e de que nunca deva levar mais que 20 segundos. Tais restrições de desempenho determinam prioridades adicionais nos atributos que são candidatos a caminhos de acesso. Os atributos de seleção utilizados pelas consultas e pelas transações com restrições de tempo se tornam candidatos de mais alta prioridade para estruturas de acesso primárias.

D. Análise das Frequências Esperadas de Operações de Atualização. Um número mínimo de caminhos de acesso deve ser especificado para um arquivo que é atualizado frequentemente porque a própria atualização das rotas de acesso diminui a velocidade das operações de atualização.

E. Análise das Restrições de Unicidade de Atributos. Os caminhos de acesso devem ser especificados para todos os atributos candidatos a chave — ou conjuntos de atributos —, que tanto são a chave primária quanto os atributos com a restrição de serem únicos. A existência de um índice (ou outro caminho de acesso) torna suficiente pesquisar apenas no índice quando da verificação dessa restrição, uma vez que todos os valores do atributo existirão nos nós folhas do índice.

Uma vez compiladas as informações anteriores, podemos nos voltar para as decisões de projeto físico do banco de dados, as quais consistem principalmente em decidir as estruturas de armazenamento e os caminhos de acesso para os arquivos do banco de dados.

16.1.2 Decisões de Projeto Físico de Bancos de Dados

A maioria dos sistemas relacionais representa cada relação base como um arquivo físico no banco de dados. As opções de caminho de acesso incluem a especificação do tipo do arquivo para cada relação e dos atributos sobre quais índices devem ser definidos. No máximo um dos índices para cada arquivo pode ser um índice primário ou *clustering*. Podem ser criados quaisquer números de índices secundários adicionais.

Decisões de Projeto sobre Indexação. Os atributos cujos valores são necessários em condições de igualdade ou de faixas (*range*) (operação de seleção) e aqueles que são chave ou que participam de condições de junção (operação de junção) necessitam de caminhos de acesso.

O desempenho das consultas depende muito de quais índices ou esquemas de *hash* existem para acelerar o processamento de seleções e junções. Porém, durante as operações de inclusão, exclusão ou atualização, a existência de índices acrescenta uma sobrecarga que precisa se justificar pelo ganho em eficiência por meio da aceleração das consultas e das transações. As decisões de projeto físico referentes à indexação caem nas seguintes categorias:

- 2 O leitor deve rever os vários tipos de índices descritos na Seção 13.1. Para uma compreensão mais clara dessa discussão, também é útil se familiarizar com os algoritmos para o processamento de consultas discutidos no Capítulo 15.

1. *Decisão de indexar segundo um atributo.* O atributo deve ser uma chave, ou deve existir alguma consulta que usa o atributo tanto em uma condição de seleção (igualdade ou faixa de valores) quanto em uma junção. Um fator a favor do estabelecimento de muitos índices é que algumas consultas podem ser processadas apenas por meio da varredura dos índices, sem a recuperação de qualquer dado.
 2. *Qual(is) atributo ou atributos se deve indexar.* Um índice pode ser construído segundo um ou múltiplos atributos. Se múltiplos atributos de uma relação são envolvidos em conjunto em diversas consultas [por exemplo, (num_esti 1 o_vestua-rio, cor) em um banco de dados de estoque de roupas], um índice de múltiplos atributos é justificado. A ordem dos atributos dentro de um índice de múltiplos atributos deve corresponder às consultas. Por exemplo, o índice acima supõe que as consultas seriam baseadas em uma ordem de cores dentro de um num_estilo_vestuário em vez do contrário.
 3. *Decisão de estabelecer um índice clustering.* No máximo um índice por tabela pode ser um índice primário ou *clustering* porque isso implica que o arquivo seja fisicamente ordenado segundo aquele atributo. Na maioria dos SGBDRs, especifica-se com a palavra-chave CLUSTER. (Se o atributo for uma chave, um índice primário é criado, enquanto um índice *clustering* é criado se o atributo não for uma chave.) Se uma tabela exigir diversos índices, a decisão sobre qual deles deveria ser um índice *clustering* dependerá da necessidade de manter a tabela ordenada segundo aquele atributo. As consultas de faixa se beneficiam bastante do *clustering*. Se diversos atributos exigem consultas de faixas, os benefícios relacionados devem ser avaliados antes de decidir qual dos atributos será utilizado no *clustering*. Se uma consulta deve ser respondida apenas pela busca em um índice (sem a recuperação de registros de dados), o índice correspondente *não* deverá ser o de *clustering*, uma vez que o principal benefício do *clustering* é obtido quando se recuperam os próprios registros.
 - 4- *Decisão de utilizar um índice hash em vez de um índice de árvore.* Em geral, os SGBDRs usam as árvores-B para a indexação. Entretanto, ISAM e índices *hash* também são disponibilizados em alguns sistemas (Capítulo 14). As árvores-B permitem consultas de igualdade e de faixas baseadas no atributo utilizado como chave de busca. Os índices *hash* funcionam bem com condições de igualdade, particularmente durante as junções para encontrar um (ou uns) registro(s) correspondente(s).
 5. *Decisão de utilizar hashing dinâmico para o arquivo.* Para arquivos que são muito voláteis — ou seja, aqueles que crescem e encolhem continuamente —, um dos esquemas de *hashing* dinâmico visto na Seção 13.9 seria adequado. Atualmente, eles não são oferecidos pela maioria dos SGBDRs comerciais.
- Desnormalização como Decisão de Projeto para Acelerar Consultas. O objetivo final durante a normalização (capítulos 10 e 11) é separar em tabelas os atributos logicamente relacionados para minimizar redundâncias e, desse modo, evitar as anomalias de atualização que levam a uma sobrecarga adicional de processamento para manter a consistência do banco de dados. Esse objetivo às vezes é sacrificado em função de uma execução mais rápida de consultas e transações que ocorrem freqüentemente. O processo de armazenar o projeto lógico do banco de dados (o qual pode estar em FNBC ou 4FN) em uma forma normal mais fraca, digamos 2FN ou IFN, é chamado de desnormalização. Normalmente o projetista adiciona a uma tabela os atributos que são necessários para responder a consultas ou produzir relatórios, de forma que uma junção com uma outra tabela, a qual contém o atributo recém-adicionado, seja evitada. Isso introduz novamente uma dependência funcional parcial ou uma dependência transitiva na tabela, criando os problemas de redundância associados (Capítulo 10).
- Outras formas de desnormalização são o armazenamento de tabelas adicionais para manter as dependências funcionais originais que são perdidas durante a decomposição FNBC. Por exemplo, a Figura 10.13 mostrou a relação ENSINA(ALUNO, CURSO, INSTRUTOR) com as dependências funcionais { {ALUNO, CURSO} → INSTRUTOR, INSTRUTOR → CURSO}. Uma decomposição sem perdas de ENSINA em E1(ALUNO, INSTRUTOR) e em E2(INSTRUTOR, CURSO) *não* permite que consultas do tipo 'qual curso fez o aluno Smith com o instrutor Navathe' sejam respondidas sem a junção de E1 e E2. Portanto, armazenar E1, E2 e ENSINA pode ser uma solução possível, o que reduz o projeto da FNBC para 3FN. Aqui, ENSINA é uma junção materializada das outras duas tabelas, representando uma redundância extrema. Quaisquer atualizações em E1 e E2 deveriam ser aplicadas a ENSINA. Uma estratégia alternativa é considerar E1 e E2 como tabelas-base atualizáveis, e ENSINA pode ser criada como uma visão.

16.2 Uma Visão Geral da Sintonização de Banco de Dados em Sistemas Relacionais

Após um banco de dados ser desenvolvido e estar em operação, o uso real das aplicações, das transações, das consultas e das visões revela fatores e áreas de problemas que podem não ter sido considerados durante o projeto físico inicial. As informa-

1 6.2 Uma Visão Geral da Sintonização de Banco de Dados em Sistemas Relacionais 389

ções de entrada para o projeto físico listadas na Seção 16.1.1 podem ser revisadas por meio da coleta de estatísticas reais sobre os padrões de uso. A utilização dos recursos, bem como o processamento interno do SGBD — tais como a otimização de consultas —, podem ser monitorados para revelar gargalos, tais como a disputa pelos mesmos dados ou dispositivos. Os volumes de atividades e os tamanhos dos dados podem ser mais bem estimados. Portanto, é necessário monitorar e revisar o projeto físico do banco de dados constantemente. Os objetivos da sintonização são os seguintes:

- Fazer com que as aplicações sejam executadas mais rapidamente.
- Diminuir o tempo de resposta de consultas/transações.
- Melhorar o desempenho geral das transações.

A linha divisória entre o projeto físico e a sintonização é muito tênue. As mesmas decisões de projeto que discutimos na Seção 16.1.3 são revisitadas durante a fase de sintonização, que é um ajuste continuado do projeto. Abaixo damos apenas uma breve visão geral do processo de sintonização. As informações de entrada para o processo de sintonização incluem estatísticas relacionadas aos fatores mencionados na Seção 16.1.1. Em particular, os SGBDs podem coletar internamente as seguintes estatísticas:

- Tamanhos de tabelas individuais.
- Número de valores distintos em uma coluna.
- Número de vezes que uma consulta ou transação em particular é submetida/executada em um intervalo de tempo.
- Os tempos necessários para as diferentes fases do processamento de consultas e transações (para um dado conjunto de consultas e transações).

Essas e outras estatísticas criam um perfil do conteúdo e do uso do banco de dados. Outras informações obtidas a partir do monitoramento das atividades do sistema de banco de dados incluem as seguintes:

- *Estatísticas de armazenamento*: Dados a respeito da alocação de armazenamento para espaço de tabelas (*tablespaces*), espaço de índices (*indexspaces*) e portas de *buffer*.
- *Estatísticas de desempenho de entrada/saída e de dispositivos*: Atividade total de leitura/escrita (paginação) na extensão do disco e nas posições mais acessadas (*hot spots*) do disco.
- *Estatísticas do processamento de consultas/transações*: Tempos de execução de consultas e transações, tempos de otimização durante a otimização de consultas.
- *Estatísticas relacionadas a bloqueios/registo de log*: Taxas de definição de diferentes tipos de bloqueios, taxas de desempenho de transações e registros de *log* de atividades.
- *Estatísticas de índices*: Número de níveis em um índice, número de páginas folha não adjacentes etc.

Muitas das estatísticas acima se referem às transações, ao controle de concorrência e à recuperação, os quais serão vistos a partir do Capítulo 17 até o Capítulo 19. A sintonização de um banco de dados envolve tratar os seguintes tipos de problema:

- Como evitar excessivas disputas por bloqueios, aumentando, desse modo, a concorrência entre as transações.
- Como minimizar a sobrecarga de registrar *logs* e o armazenamento desnecessário de dados.
- Como otimizar o tamanho do *buffer* e o escalonamento de processos.
- Como alocar recursos tais como discos, RAM e processos para uma utilização mais eficiente.

A maioria dos problemas mencionados anteriormente pode ser resolvida por meio do ajuste apropriado de parâmetros físicos do SGBD, da alteração das configurações de dispositivos, da alteração de parâmetros do sistema operacional e de outras atividades similares. As soluções normalmente estão ligadas a sistemas específicos. Normalmente, os DBAs são treinados para tratar esses problemas de sintonização em SGBDs específicos. Abaixo, analisaremos brevemente a sintonização de várias decisões de projeto físico de bancos de dados.

16.2.1 Sintonização de índices

A escolha inicial de índices pode precisar de uma revisão pelas seguintes razões:

- Certas consultas podem demorar demais para ser executadas por conta da ausência de um índice.
- Certos índices podem, absolutamente, não ser utilizados.

3 Os leitores interessados podem consultar uma análise detalhada sobre sintonização em Shasha (1992).

4 O leitor deverá ler os capítulos 17, 18 e 19 para obter explicações sobre esses termos.

390 Capítulo 16 Projeto e Sintonização (*Tuning*) de Bancos de Dados na Prática

• Certos índices podem estar causando sobrecarga excessiva porque são baseados em um atributo que sofre freqüentes alterações. A maioria dos SGBDs possui um comando ou um meio de rastreamento (*trace facility*) que pode ser usado pelo DBA para solicitar que o sistema mostre como uma consulta foi executada — quais operações foram realizadas e em qual ordem e quais estruturas de acesso secundário foram utilizadas. Por meio da análise desses planos de execução, é possível diagnosticar as causas dos problemas acima. Alguns índices podem ser excluídos e alguns novos índices podem ser criados com base na análise de sintonização.

O objetivo da sintonização é avaliar dinamicamente os requisitos, os quais às vezes variam sazonalmente ou durante diferentes períodos do mês ou da semana, e reorganizar os índices para proporcionar melhor desempenho geral. A exclusão e a construção de novos índices são uma sobrecarga que pode ser justificada em função das melhorias de desempenho. A atualização de uma tabela geralmente é suspensa enquanto um índice estiver sendo excluído ou criado; deve-se contabilizar essa perda de serviço. Além da exclusão ou da criação de índices e a mudança a partir de um índice que não é *clustering* para um índice *clustering* e vice-versa, a **reconstrução** do índice pode melhorar o desempenho. A maioria dos SGBDRs usa árvores-B em índices. Se houver muitas exclusões na chave do índice, páginas do índice podem ter espaço desperdiçado, o que pode ser recuperado durante a operação de reconstrução. De forma similar, muitas inclusões podem causar *overflow* em um índice *clustering* que afetam o desempenho. A reconstrução de um índice *clustering* equivale a reorganizar a tabela ordenada inteira segundo aquela chave.

As opções disponíveis para a indexação e a maneira como os índices são definidos, criados e reorganizados variam de sistema para sistema. Apenas para ilustração, considere os índices esparsos e densos do Capítulo 14. Os índices esparsos possuem um ponteiro de índice para cada página (bloco de disco) de um arquivo de dados; os índices densos possuem um ponteiro de índice para cada registro. O Sybase prove índices *clustering* como índices esparsos na forma de árvores-B, enquanto o INGRES prove índices *clustering* esparsos como arquivos ISAM e índices *clustering* densos como árvores-B. Em algumas versões da Oracle e do DB2, a opção de configuração de um índice *clustering* está limitada a um índice denso (com muito mais entradas de índice), e o DBA tem de trabalhar com essa limitação.

16.2.2 Sintonização do Projeto de Banco de Dados

Já discutimos na Seção 16.1.2 a necessidade de uma possível desnormalização, que significa abandonar a decisão de manter todas as tabelas como relações FNBC. Se um dado projeto físico de banco de dados não atinge os objetivos esperados, poderemos voltar ao projeto lógico do banco de dados, fazer ajustes no esquema lógico e remapeá-lo num novo conjunto de tabelas e índices físicos. Conforme destacamos, todo o projeto do banco de dados deve ser direcionado pelos requisitos de processamento e pelos requisitos dos dados. Se os requisitos de processamento se alteram dinamicamente, o projeto precisa responder por meio de alterações no esquema conceitual, se necessário, e refletir aquelas mudanças no esquema lógico e no projeto físico. Essas alterações podem ser da seguinte natureza:

- Tabelas existentes podem ser juntadas (desnormalizadas) porque certos atributos de duas ou mais tabelas são freqüentemente necessários em conjunto: isso reduz o nível de normalização de FNBC para 3FN, 2FN ou IFN.
 - Para dado conjunto de tabelas, pode haver escolhas de projeto alternativas, todas as quais obtêm a 3FN ou a FNBC. Uma pode ser substituída pela outra.
 - Uma relação da forma R(K, A, B, C, D, ...), onde K é um conjunto de atributos-chave — que está na FNBC —, pode ser armazenada em múltiplas tabelas que também estão na FNBC — por exemplo, R1(K, A, B), R2(K, C, D) e R3(K, ...) — por meio da replicação da chave K em cada tabela, à qual reúne conjuntos de atributos que são acessados em conjunto. Por exemplo, a tabela EMPREGADO(SSN, Nome, Telefone, Formação, Salário) pode ser dividida em duas tabelas EMP1(SSN, Nome, Telefone) e EMP2(SSN, Formação, Salário). Se a tabela original possuía um número muito grande de linhas (digamos cem mil) e as consultas sobre informações de números de telefone e salários forem totalmente distintas, essa separação de tabelas pode funcionar melhor. Isso também é chamado **partição vertical**.
 - Atributo(s) de uma tabela pode(m) ser repetido(s) em uma outra mesmo que isso crie redundância e uma anomalia potencial. Por exemplo, NomePeca pode ser replicada em tabelas onde quer que NumeroPeca apareça (como uma chave
- 5 Observe que 3FN e 2FN são voltadas para diferentes tipos de problemas de dependência, os quais são independentes entre si; por isso a ordem de normalização (ou desnormalização) entre elas é arbitrária.

1 6.2 Uma Visão Gera! da Sintonização de Banco de Dados em Sistemas Relacionais 391

estrangeira), mas pode haver uma tabela mestre chamada MESTRE_PECA(NumeroPeca, NomePeca), onde se garanta que NomePeca esteja atualizado. • Da mesma forma que a partição vertical divide uma tabela verticalmente em múltiplas tabelas, a **partição horizontal** toma fatias horizontais de uma tabela e as armazena como tabelas distintas. Por exemplo, os dados de vendas de produtos podem ser separados em dez tabelas baseadas em dez linhas de produtos. Cada tabela possui o mesmo conjunto de colunas (atributos), porém, contém um conjunto distinto de produtos (tuplas). Se uma consulta ou uma transação se aplica a todos os dados de produtos, ela talvez precise ser executada em todas as tabelas e os resultados podem precisar ser combinados. Esses tipos de ajustes, projetados para atender a consultas e transações de grandes volumes, com ou sem o sacrifício das formas normais, são comuns na prática.

16.2.3 Sintonização de Consultas

Já vimos como o desempenho de consultas é dependente da seleção apropriada de índices e como eles precisam ser sintonizados após a análise das consultas que dão um desempenho ruim por meio do uso de comandos do SGBDR que mostram o plano de execução da consulta. Há principalmente duas indicações que sugerem que a sintonização da consulta pode ser necessária:

1. Uma consulta resulta em muitos acessos a disco (por exemplo, uma consulta de correspondência exata varre uma tabela inteira).
2. Um plano de consulta mostra que os índices relevantes não estão sendo utilizados.

Vejam algumas situações típicas que indicam a necessidade de sintonização de consultas:

1. Muitos otimizadores de consulta não usam índices na presença de expressões aritméticas (tais como SALÁRIO/365 > 10,50), de comparações numéricas de atributos de diferentes tamanhos e níveis de precisão (tais como QTDDE_A = QTDDE_B, onde QTDDE_A é do tipo INTEGER e QTDDE_B é do tipo SMALLINTEGER), comparações COM NULL (tais como DATANASC is NULL), e comparações com *substrings* (tais como UNOME LIKE "%MANN").

2. Frequentemente os índices não são usados em consultas aninhadas usando IN; por exemplo, a consulta

```
SELECT SSN FROM EMPREGADO
```

```
WHERE NRD IN (SELECT NUMEROD FROM DEPARTAMENTO WHERE GERSSN = '333445555');
```

pode não usar o índice para NRD em EMPREGADO, enquanto o uso de NRD = NUMEROD na cláusula WHERE com uma consulta em um único bloco pode gerar o índice a ser usado.

3. Algumas cláusulas DISTINCT podem ser redundantes e podem ser evitadas sem a modificação do resultado. Um DISTINCT frequentemente causa uma operação de ordenação e deve ser evitado sempre que possível.

4. O uso desnecessário de tabelas de resultado temporário pode ser evitado por meio da aglutinação de múltiplas consultas em uma única consulta, *a menos* que a relação temporária seja necessária para algum processamento intermediário.

5. Em algumas situações que envolvem o uso de consultas correlatas, os temporários são úteis. Considere a consulta:

```
SELECT SSN
```

```
FROM EMPREGADO E
```

```
WHERE SALÁRIO = SELECT MAX (SALÁRIO)
```

```
FROM EMPREGADO AS G
```

```
WHERE G.NRD = E.NRD;
```

Ela possui o perigo potencial de pesquisar toda a tabela EMPREGADO G interna para *cada* tupla da tabela EMPREGADO E externa. Para torná-la mais eficiente, ela pode ser quebrada em duas consultas, na qual a primeira apenas calcula o máximo salário em cada departamento conforme segue:

```
SELECT MAX (SALÁRIO) AS MAIORSALARIO, NRD INTO TEMP FROM EMPREGADO GROUP BY NRD;
```

```
SELECT SSN
```

```
FROM EMPREGADO, TEMP
```

```
WHERE SALÁRIO = MAIORSALARIO AND EMPREGADO.NRD = TEMP.NRD;
```

392 Capítulo 16 Projeto e Sintonização (*Tuning*) de Bancos de Dados na Prática

6. Se múltiplas opções de condições de junção são possíveis, escolha uma que use um índice *clustering* e evite as que contenham comparações de cadeias de caracteres. Por exemplo, supondo que o atributo NOME é uma chave candidata em EMPREGADO e ALUNO, é melhor usar EMPREGADO.SSN = ALUNO.SSN como uma condição de junção em vez de EMPREGADO.NOME = ALUNO.NOME se SSN possuir um índice *clustering* em uma ou ambas as tabelas.

7. Uma idiossincrasia dos otimizadores de consulta é que a ordem das tabelas na cláusula FROM pode afetar o processamento de junções. Se esse for o caso, pode-se precisar alterar essa ordem de forma que a menor das duas relações seja varrida e a maior relação seja usada com um índice adequado.

8. Alguns otimizadores de consulta têm um desempenho pior em consultas aninhadas em comparação com suas consultas não aninhadas correspondentes. Há quatro tipos de consultas aninhadas:

- Subconsultas não correlatas com agregações na consulta interna.
- Subconsultas não correlatas sem agregações.
- Subconsultas correlatas com agregações na consulta interna.
- Subconsultas correlatas sem agregações.

Dos quatro tipos acima, o primeiro geralmente não apresenta problemas, uma vez que a maioria dos otimizadores de consulta avalia a consulta interna uma vez. Entretanto, para uma consulta do segundo tipo, tal como o exemplo no item 2 acima, a maioria dos otimizadores pode não utilizar um índice para NRD em EMPREGADO. Os mesmos otimizadores podem fazê-lo se a consulta for escrita como uma consulta não aninhada. A transformação de subconsultas correlatas pode envolver o estabelecimento de tabelas temporárias. Exemplos detalhados estão fora do nosso objetivo neste trabalho.

9. Finalmente, muitas aplicações são baseadas em visões que definem os dados de interesse para aquelas aplicações. Às vezes, essas visões se tornam excessivas, porque uma consulta pode ser aplicada diretamente sobre uma tabela base em vez de ser realizada sobre uma visão que é definida por uma junção.

16.2.4 Orientações Adicionais para a Sintonização de Consultas

Técnicas adicionais para a melhoria de consultas se aplicam em certas situações:

1. Uma consulta com múltiplas condições de seleção que são conectadas por meio de OR podem não estar sinalizando ao otimizador de consultas para usar um índice. Tal consulta pode ser dividida e expressa como uma união de consultas, cada uma com uma condição para um atributo que tenha um índice a ser utilizado. Por exemplo:

```
SELECT PNAME, UNOME, SALÁRIO, IDADE
FROM EMPREGADO
```

```
WHERE IDADE > 45 AND SALÁRIO < 50.000;
```

pode ser executada utilizando uma varredura sequencial, que dá um desempenho ruim. Sua divisão como:

```
SELECT PNAME, UNOME, SALÁRIO, IDADE FROM EMPREGADO WHERE IDADE > 45;
```

```
UNION
```

```
SELECT PNAME, UNOME, SALÁRIO, IDADE
```

```
FROM EMPREGADO
```

```
WHERE SALÁRIO < 50.000;
```

pode utilizar índices para IDADE bem como para SALÁRIO.

2. Para ajudar a acelerar uma consulta, podemos tentar as seguintes transformações:

- A condição NOT pode ser transformada em uma expressão afirmativa.
- Blocos SELECT embutidos utilizando IN, = ALL, = ANY e = SOME podem ser substituídos por junções.
- Se uma junção de igualdade for definida entre duas tabelas, o predicado de faixa (condição de seleção) para o atributo de junção definido em uma tabela pode ser repetido para a outra tabela.
- Condições WHERE podem ser reescritas para utilizar os índices para múltiplas colunas. Por exemplo:

67

Para mais detalhes, veja Shasha (1992).

Modificamos o esquema e usamos IDADE em EMPREGADO em vez de DATANASC.

16.3 Resumo 393

```
SELECT NUM_REGIAO, TIPO_PRODUTO, MES, VENDAS
FROM ESTATISTICAJENDAS
WHERE NUM_REGIAO = 3 AND ((TIPO_PRODUTO BETWEEN 1 AND 3) OR (TIPO_PRODUTO
BETWEEN 8 AND 10));
```

pode utilizar um índice apenas para NUM_REGIAO e pesquisar todas as páginas folhas do índice para uma correspondência em TIPO_PRODUTO. Em vez disso, usando

```
SELECT NUM_REGIAO, TIPO_PRODUTO, MES, VENDAS
FROM ESTATISTICAJENDAS
WHERE (NUM_REGIAO = 3 AND (TIPO_PRODUTO BETWEEN 1 AND 3)) OR (NUM_REGIAO =
3 AND (TIPO_PRODUTO BETWEEN 8 AND 10));
```

pode utilizar um índice composto para (NUM_REGIAO, TIPO_PRODUTO) e funcionar muito mais eficientemente.

Nesta seção cobrimos a maioria das oportunidades comuns nas quais a ineficiência de uma consulta pode ser corrigida por alguma ação corretiva simples, tal como o uso de um arquivo temporário, evitando certas construções ou o uso de visões. Os problemas e as soluções dependerão do trabalho de um otimizador de consulta dentro de um SGBDR. Há literatura detalhada com orientações sobre a sintonização de banco de dados para a administração de banco de dados produzidos pelos fornecedores de SGBDR no formato de manuais.

16.3 RESUMO

Neste capítulo discutimos os fatores que afetam as decisões de projeto físico do banco de dados e fornecemos orientações para a escolha entre as alternativas de projeto físico. Analisamos alterações no projeto lógico, modificações na indexação e alterações em consultas como parte da sintonização de banco de dados.

Questões para Revisão

- 16.1 Quais são os fatores importantes que influenciam o projeto físico de banco de dados?
- 16.2 Discuta as decisões tomadas durante o projeto físico de banco de dados.
- 16.3 Discuta as orientações para o projeto físico de banco de dados em SGBDRs.
- 16.4 Discuta os tipos de modificações que podem ser aplicadas a um projeto lógico de banco de dados de um banco de dados relacional.
- 16.5 Sob quais situações a desnormalização de um esquema de banco de dados seria utilizada? Dê exemplos de desnormalização.
- 16.6 Discuta a sintonização de índices para bancos de dados relacionais.
- 16.7 Discuta as considerações para a reavaliação e a modificação de consultas SQL.
- 16.8 Ilustre os tipos de alterações em consultas SQL que podem valer a pena ser considerados para a melhoria do desempenho durante a sintonização do banco de dados.
- 16.9 Quais as funções que as ferramentas típicas de projeto de banco de dados provêm?

Bibliografia Seleccionada

Wiederhold (1986) cobre todas as fases do projeto de banco de dados, com ênfase no projeto físico. O'Neil (1994) faz uma análise detalhada do projeto físico e de questões de transação referentes a SGBDRs comerciais. Navathe e Kerschberg (1986) discutem todas as fases do projeto de banco de dados e destacam o papel dos dicionários de dados. Rozen e Shasha (1991) e Carlis e March (1984) apresentam diferentes modelos para o problema de projeto físico de banco de dados.

5

Conceitos de Processamento de Transações

17

Introdução aos Conceitos e à Teoria do Processamento de Transações

O conceito de transação fornece um mecanismo para descrição de unidades lógicas de processamento de banco de dados. Sistemas de processamento de transação são sistemas com grandes bancos de dados e centenas de usuários executando transações concorrentes no banco de dados. Exemplos desses tipos de sistemas englobam os sistemas para reservas de passagens, bancos, processamento de cartões de crédito, mercados de ações, entregas de supermercados, entre outros. Eles exigem alta disponibilidade e baixo tempo de resposta para as centenas de usuários concorrentes. Neste capítulo apresentaremos os conceitos necessários aos sistemas de processamento de transação. Definiremos o conceito de uma transação usado para representar uma unidade lógica de processamento de banco de dados, que deve ser completo e integral para garantir precisão. Analisaremos o problema do controle de concorrência, quando diversas transações, submetidas por vários usuários, interferirem nas outras, produzindo resultados incorretos. Discutiremos também a restauração (ou recuperação) de falhas de transação.

A Seção 17.1 discutirá informalmente por que o controle de concorrência e a restauração são necessários em um sistema de banco de dados. A Seção 17.2 introduzirá o conceito de transação e discutirá conceitos adicionais relacionados ao processamento de transação em sistemas de banco de dados. A Seção 17.3 apresentará os conceitos de atomicidade, preservação de consistência, isolamento e durabilidade (ou persistência) — chamados de propriedades ACID —, que são considerados desejáveis em transações. A Seção 17.4 introduzirá o conceito de planos (ou histórico) de execução de transações e caracterizará a restaurabilidade (ou capacidade de restauração) de planos. A Seção 17.5 discutirá o conceito de serialidade em execuções de transação concorrente, que pode ser usado para definir seqüências de execução (ou planos) corretas com transações concorrentes. A Seção 17.6 apresentará as facilidades que dão suporte ao conceito de transação em SQL.

Os dois capítulos subsequentes detalharão as técnicas usadas no processamento de transações. O Capítulo 18 descreverá as técnicas básicas para controle de concorrência e o Capítulo 19, uma visão das técnicas de restauração (ou recuperação).

17.1 INTRODUÇÃO AO PROCESSAMENTO DE TRANSAÇÕES

Nesta seção introduziremos, formalmente, os conceitos de execução concorrente de transações e restauração de falhas em transações. A Seção 17.1.1 comparará sistemas de banco de dados monousuário e multiusuário, e demonstrará como a execução concorrente de transações se comporta em um sistema multiusuário. A Seção 17.1.2 definirá o conceito de transação e apresentará um modelo simples de execução de transações, baseado em operações de leitura e gravação em banco de dados, usado para formalizar os conceitos de controle de concorrência e restauração. A Seção 17.1.3 mostrará, por meio de exemplos informais, por que são necessárias técnicas para controle de concorrência em sistemas multiusuários. Finalmente, a Seção

Optou-se por *restauração* para tradução da palavra inglesa *recovery*, a fim de se evitar a repetição do termo *recuperação* usado para a palavra *retrieval* (recuperar, puxar, trazer dados do banco de dados). (N. de **R.T.**) 398

17.1.4 discutirá por que são necessárias técnicas que possibilitem restauração de falhas, discutindo os diferentes meios pelos quais transações em execução podem falhar.

17.1.1 Sistemas Monousuários Versus Multiusuários

Um dos critérios de classificação de um sistema de banco de dados refere-se ao número de usuários que podem usar o sistema **concorrentemente** — isto é, ao mesmo tempo. Um SGBD será **monousuário** se somente um usuário de cada vez puder usar o sistema, e será **multiusuário** se muitos usuários puderem usá-lo — e, portanto, acessar o banco de dados — concorrentemente. SGBDs monousuários estão mais restritos aos sistemas de computação pessoal; a maioria dos demais SGBDs é multiusuário; Por exemplo, um sistema de reservas de uma empresa aérea é usado por centenas de agências de viagem e balcões de reserva concorrentemente. Sistemas bancários, seguradoras, casas de câmbio, supermercados, entre outros, também são operados por muitos usuários, que submetem transações concorrentes ao sistema.

Diversos usuários podem ter acesso aos bancos de dados — e usar os sistemas de computador — simultaneamente por causa do conceito de **multiprogramação**, que permite que o computador execute diversos programas — ou **processos** — ao mesmo tempo. Se houver apenas uma unidade central de processamento (CPU), ela poderá executar de fato, no máximo, um processo de cada vez. Entretanto, **sistemas operacionais com multiprogramação** executam alguns comandos de um processo depois suspendem esse processo e executam alguns comandos do próximo processo, e assim por diante. O processo será retomado no ponto em que foi suspenso quando retornar à CPU. Portanto, a execução concorrente de processos é, na verdade, intercalada, como ilustrado na Figura 17.1, que apresenta dois processos, A e B, executados concorrentemente de modo intercalado. A intercalação possibilita a ocupação da CPU, enquanto um processo executa uma operação de entrada ou saída (I/O), como ler um bloco de um disco. A CPU será chaveada para executar outro processo em vez de permanecer ociosa durante esse tempo de I/O. A intercalação também impede que um processo longo retarde outros processos.

CPUi CPUz

«i

L

Tempo

FIGURA 17.1 Processamento intercalado versus processamento paralelo de transações concorrentes.

Se o sistema de computador possuir vários processadores (CPUs), será possível o **processamento paralelo** de diversos processos, como ilustrado na Figura 17.1 pelos processos C e D. A maior parte da teoria relacionada ao controle de concorrência em bancos de dados foi desenvolvida em termos de **concorrência intercalada**, assim, no restante deste capítulo, assumiremos esse modelo. Em um SGBD multiusuário, os itens de dados armazenados são os recursos primários que devem ser acessados concorrentemente por usuários interativos ou programas de aplicação, que estão constantemente recuperando informações e modificando o banco de dados.

17.1.2 Transações, Operações de Leitura e Escrita e *Buffers* de SGBD

Uma **transação** é um programa em execução que forma uma unidade lógica de processamento no banco de dados. Uma transação inclui uma ou mais operações de acesso ao banco de dados — englobam operações de inserção, exclusão, alteração ou recuperação. As operações de banco de dados que formam uma transação podem estar embutidas em um programa de aplicação ou podem ser especificadas interativamente, via uma linguagem de consulta de alto nível, tal como a SQL. Um meio para especificar os limites de uma transação é estabelecer explicitamente declarações de **início de transação** e **fim de transação** em um programa de aplicação; nesse caso, todas as operações de acesso ao banco de dados, entre as duas, serão consideradas parte da transação. Um programa de aplicação pode conter mais de uma transação — se ele contiver diversos limites de transações.

17.1 Introdução ao Processamento de Transações 399

Se as operações em um banco de dados de uma transação não atualizarem o banco de dados, apenas recuperarem os dados, a transação será chamada de **transação de leitura (read-only)**.

O modelo de banco de dados que será usado para explicar os conceitos de processamento de transações é muito simplificado. O banco de **dados** é basicamente representado por uma coleção de **itens** de dados denominados. O tamanho do item de dado será sua **granularidade**, que pode ser um campo de algum registro do banco de dados ou pode ser uma unidade maior, como um registro inteiro ou mesmo um bloco de disco, mas os conceitos que discutiremos são independentes da granularidade do item de dado.

Usando esse modelo de banco de dados simplificado, as operações básicas de acesso ao banco de dados que uma transação pode conter são as seguintes:

- **ler_item(X)**: Lê um item do banco de dados, chamado X, em uma variável de programa. Para simplificar nossa notação, assumiremos que a variável de programa também seja chamada X.
- **gravar_item(X)**: Grava o valor de uma variável de programa X em um item de banco de dados chamado X.

Como vimos no Capítulo 13, a unidade básica de transferência de dados do disco para a memória principal é o bloco. Executar um comando ler_item(X) engloba os seguintes passos:

1. Encontrar o endereço do bloco de disco que contém o item X.
2. Copiar esse bloco para um *buffer* na memória principal (se o bloco de disco ainda não estiver em algum *buffer* da memória principal).
3. Copiar o item X do *buffer* para a variável de programa chamada X.

Executar um comando gravar_item(X) compreende os seguintes passos:

1. Encontrar o endereço do bloco de disco que contém o item X.
2. Copiar esse bloco de disco para um *buffer* na memória principal (se ele ainda não estiver em algum *buffer* da memória principal).
3. Copiar o item X de uma variável de programa chamada X para seu local correto no *buffer*.

Armazenar o bloco atualizado no *buffer* de volta para o disco (imediatamente ou em algum momento posterior).

O passo 4 é o que realmente atualizará o banco de dados no disco. Em alguns casos, se houver outras mudanças a serem feitas no *buffer*, ele não será imediatamente armazenado no disco. Normalmente a decisão de quando armazenar um bloco de disco alterado, que está no *buffer* da memória principal, é tomada pelo administrador de restauração do SGBD em conjunto com o sistema operacional. O SGBD mantém, em geral, alguns *buffers* na memória principal que contêm os blocos de disco do banco de dados com os itens que estiverem sendo processados. Quando esses *buffers* estiverem todos ocupados, e outros blocos do banco de dados precisarem ser copiados na memória, alguma política de substituição de *buffer* será utilizada para escolher qual entre os *buffers* correntes será substituído. Se o *buffer* escolhido tiver sido modificado, ele deverá ser gravado no disco antes de ser reutilizado.

Uma transação compreende operações ler_item e gravar_item para acessos e atualizações do banco de dados. A Figura 17.2 mostrará exemplos de duas transações muito simples. O **conjunto_leitura** de uma transação será o conjunto de todos os itens que a transação ler, e o **conjunto_gravação** será o conjunto de todos os itens que a transação gravar. Por exemplo, o conjunto_leitura de T_1 na Figura 17.2, é {X, Y}, e seu conjunto_gravação também é {X, Y}.

(a) T_1	(b) T_2
leMtem (X);	escreveMtem (X);
$X := X - N$;	$X := X + M$;
escrever_item (X);	escreveMtem (X);
leMtem (Y);	
$Y := Y + N$;	
escreveMtem (Y);	

FIGURA 17.2 Duas transações simples, (a) Transação T_1 , (b) Transação T_2 .

Tanto o controle de concorrência quanto os mecanismos para restauração estão relacionados principalmente com os comandos de acesso a banco de dados de uma transação. Transações submetidas por vários usuários podem ser executadas concorrentemente e podem acessar e atualizar os mesmos itens no banco de dados. Se essa execução for descontrolada, poderão

1 Não discutiremos aqui as políticas para substituição de *buffers* da maneira como elas são normalmente vistas em livros de sistemas operacionais.

ocorrer problemas de inconsistência no banco de dados. Na próxima seção, veremos informalmente alguns desses possíveis problemas.

17.1.3 Por que o Controle de Concorrência é Necessário

Diversos problemas podem ocorrer quando transações concorrentes são executadas de maneira descontrolada. Ilustraremos alguns desses problemas fazendo referência a um banco de dados, muito simplificado, de reservas de uma companhia aérea no qual será armazenado um registro para cada voo da companhia. Cada registro conterá, como um *item denominado de dado*, o número de poltronas reservadas para aquele voo, entre outras informações. A Figura 17.2a mostra uma transação T_1 , que *transfere* N reservas de um voo, cujo número de poltronas reservadas está em um item do banco de dados chamado X , para um outro voo, cujo número de assentos reservados está armazenado em um item de dados chamado Y . A Figura 17.2b mostra uma transação T_2 mais simples, que apenas *reserva* M assentos no mesmo voo (X) referenciado na transação T_1 . Para simplificar nosso exemplo, não mostraremos as demais etapas das transações, tais como verificar se o voo tem poltronas suficientes para atender à reserva em questão.

Quando um programa de acesso ao banco de dados for escrito, ele terá como parâmetros os números dos voos, suas datas e o número de assentos a serem reservados, portanto, o mesmo programa pode ser usado para executar muitas transações, cada uma em voos diferentes e com números diferentes de poltronas a serem reservadas. Para efeito de controle de concorrência uma transação é uma *execução particular* de um programa em uma data específica, num dado voo e com um dado número de poltronas. Nas figuras 17.2a e b, as transações T_1 e T_2 são *execuções específicas* dos programas que se referem a voos específicos cujos números de assentos estão armazenados em itens de dado X e Y no banco de dados. Discutiremos agora os tipos de problemas que podemos encontrar com essas duas transações, caso sejam executadas concorrentemente.

O Problema da Atualização Perdida. Esse problema ocorre quando duas transações que acessam os mesmos itens de banco de dados tiverem suas operações intercaladas, de forma que tornem o valor de alguns dos itens do banco de dados incorretos. Suponha que as transações T_1 e T_2 sejam submetidas aproximadamente ao mesmo tempo, e que suas operações sejam intercaladas, como mostra a Figura 17.3a; então o valor final do item X será incorreto, porque T_2 lerá o valor de X antes de T_1 mudá-lo no banco de dados, portanto, o valor atualizado resultante de T_1 será perdido. Por exemplo, se $X = 80$ no início (originalmente havia 80 reservas no voo), $N = 5$ (T_1 transfere 5 reservas de assentos do voo X para o voo Y) e $M = 4$ (T_2 reserva 4 poltronas em X), o resultado final deveria ser $X = 79$, mas, na intercalação de operações mostrada na Figura 17.3a, $X = 84$ porque a atualização feita por T_1 , que removeu cinco reservas de X , foi *perdida*.

O Problema da Atualização Temporária (ou *Dirty Read* — Leitura de Sujeira). Esse problema ocorre quando uma transação atualizar um item de banco de dados e, a seguir, falhar por alguma razão (Seção 17.1.4). O item atualizado será acessado por uma outra transação antes que ele retorne ao seu valor original. A Figura 17.3b mostra um exemplo onde T_1 atualiza o item X e falha antes de encerrar, quando o sistema deve reverter o valor original de X . Entretanto, antes que ele possa fazer isso, a transação T_2 lê o valor 'temporário' de X , que não será gravado permanentemente no banco de dados por causa da falha de T_1 . O valor do item X , que é lido por T_2 , é chamado *sujeira*, por ele ter sido criado por uma transação incompleta e ainda não efetivada — por isso esse problema é conhecido também como *problema de leitura de sujeira*.

O Problema do Sumário Incorreto. Se uma transação aplicar uma função agregada para sumário de um número de registros enquanto outras transações estiverem atualizando alguns desses registros, a função agregada deverá calcular alguns valores antes de eles serem atualizados e outros depois de feita a atualização. Por exemplo, suponha que uma transação T_3 esteja calculando o número total de reservas em todos os voos; enquanto isso, a transação T_1 é executada. Se acontecer a intercalação de operações mostrada na Figura 17.3c, o resultado de T_3 não contabilizará N , pois T_3 leu o valor de X *depois* que os N assentos foram subtraídos, mas lerá o valor de Y *antes* que esses N assentos tenham sido adicionados a ele.

Um outro problema que pode ocorrer é chamado leitura sem repetição, quando uma transação T lê um item duas vezes e o item é mudado por uma outra transação T entre essas duas leituras. Portanto, T receberá *valores diferentes* para duas leituras do mesmo item. Isso poderá ocorrer, por exemplo, se, durante uma transação para reserva na companhia aérea, um cliente consultar a disponibilidade de assento em diversos voos. Quando ele se decidir por um voo em particular, a transação então, lerá o número de assentos nesse voo uma segunda vez, antes de completar a reserva.

2 Um exemplo similar, mais comum, adota um banco de dados de um banco com uma transação de transferência de fundos de uma conta X para uma conta Y , e uma outra transação de um depósito na conta X .

17.1 Introdução ao Processamento de Transações 401

(a)

*Tempo**T*₁

ler_item(X); X:=X-N;

escrever_item(X) leUtem(y);

Y:=Y+N; escrever_item(V)

ler_item(X); X:=X+M;

escrever_item(X);

Item Xtem um valor incorreto porque a atualização feita por T² foi 'perdida' (sobrescrita)

(b)

*Tempo*A transação T₁, falha e X deverá retornar a seu valor original; enquanto isso, T₂ lê o valor 'temporário' incorreto de X.*T*₁*T*₂

Ler_item(X);

X:=X-W;

Escrever_item(X);

ler_item(X);

X:=X+M;

escrever_item(X);

ler_item(Y);

-----•

(c)

*T*₁*T*_i

sum:=0;

ler_item(A);

sum:=sum+A;

•

ler_item(X);

X:=X-N;

escreveMtem(X);

ler_item(X);

sum:=sum+X;

ler_item(V);

sum:=sum+y;

ler_item(V);

Y:=Y+N;

escrever_item(V);

T₃ lê X depois da subtração de N e lê Y antes da adição de A; o resultado é um sumário errado (sem N).**FIGURA 17.3** Alguns dos problemas que podem ocorrer quando uma execução concorrente não for controlada, (a) O problema da atualização perdida, (b) O problema da atualização temporária, (c) O problema do sumário incorreto.

17.1.4 Por que a Restauração (Recuperação) é Necessária

Se uma transação for executada por um SGBD, o sistema deverá garantir que: (1) todas as operações na transação foram completadas com sucesso e seu efeito será gravado permanentemente no banco de dados ou (2) a transação não terá nenhum efeito sobre o banco de dados ou sobre quaisquer outras transações. O SGBD não deverá permitir que algumas das operações de uma transação *T* sejam aplicadas ao banco de dados enquanto as outras operações *T* não o forem. Isso poderá acontecer se uma transação **falhar** depois de executar algumas de suas operações, mas antes de executar todas elas.

Tipos de Falhas. Geralmente as falhas são classificadas como de transação, sistema ou mídia. Há diversas razões possíveis para a falha de uma transação durante sua execução:

1. *O computador falhar (crash ou queda do sistema)*: Um erro de hardware, software ou rede ocorre em um sistema de computador durante a execução da transação. Quedas de hardware são, geralmente, falhas de mídia — por exemplo, falha da memória principal.
2. *Um erro de transação ou sistema*: Alguma operação da transação pode causar falha, como o estouro (*overflow*) de um inteiro ou uma divisão por zero. Falhas de transação também podem ocorrer por conta de valores errados de parâmetros ou por causa de um erro de lógica na programação. Além disso, o usuário pode interromper a transação durante sua execução.
3. *Erros locais ou condições de exceção detectadas pela transação*: Durante a execução da transação, podem ocorrer determinadas condições que necessitem do cancelamento da transação. Por exemplo, os dados para a transação não foram encontrados. Observe que uma condição de exceção, como um saldo insuficiente na conta do banco de dados de um banco, pode fazer com que uma transação, tal como a retirada de fundos, seja cancelada. Essa exceção seria programada na própria transação, portanto, não seria considerada uma falha.
4. *Imposição do controle de concorrência*: O método de controle de concorrência (Capítulo 18) pode optar por abortar uma transação para ser reiniciada posteriormente caso ela viole a serialização (Seção 17.5) — ou porque diversas transações estão em um estado de *deadlock* (impasse).
5. *Falha de disco*: Alguns blocos de disco podem perder seus dados por causa do mau funcionamento de uma leitura ou de uma gravação, ou por causa de um *crash* do cabeçote de leitura/escrita de um disco. Isso pode acontecer durante uma operação de leitura ou gravação da transação.
6. *Problemas físicos e catástrofes*: Referem-se a uma lista sem fim de problemas que incluem falha de energia ou de ar-condicionado, fogo, furto, sabotagem, sobregravação de dados em discos ou fitas por engano ou montagem de uma fita errada pelo operador.

Falhas dos tipos 1, 2, 3 e 4 são mais frequentes que as dos tipos 5 ou 6. Se ocorrer uma falha do tipo 1 a 4, o sistema deverá manter informações suficientes para se recuperar dessa falha. Falhas de disco ou outras falhas catastróficas, do tipo 5 ou 6, não acontecem frequentemente; se elas ocorrerem, a restauração é uma tarefa especializada. Veremos restauração (recuperação) de falhas no Capítulo 19.

O conceito de transação é fundamental para muitas das técnicas de controle de concorrência e restauração de falhas.

17.2 CONCEITOS DE TRANSAÇÃO E SISTEMA

Nesta seção discutiremos outros conceitos relevantes ao processamento de transações. A Seção 17.2.1 descreverá os vários estados em que uma transação pode estar e discutirá operações adicionais relevantes do processamento de transações. Analisaremos, na Seção 17.2.2, o *log* (registro de ocorrências) de sistema, que mantém as informações necessárias para a restauração. A Seção 17.2.3 descreverá o conceito de pontos para efetivação (*commit*) de transações e por que eles são importantes no processamento de transações.

17.2.1 Estados da Transação e Operações Adicionais

Uma transação é uma unidade atômica de trabalho que ou estará completa ou não foi realizada. Para propostas de restauração, o sistema precisa manter o controle de quando a transação inicia, termina e de suas efetivações ou interrupções (*commits* ou *aborts*) (Seção 17.2.3). Portanto, o administrador de restaurações mantém o controle das seguintes operações:

- 3 Em geral, uma transação deveria ser completamente testada, de forma a garantir que não haja *bugs* (erros de lógica de programação).
- 4 Condições de exceção, se corretamente programadas, não constituem falhas de transação.

17.2 Conceitos de Transação e Sistema 403

- **BEGIN_TRANSACTION**: Marca o início da execução da transação.
- **READ** ou **WRITE**: Especificam operações de leitura ou gravação em itens do banco de dados, que são executadas como parte de uma transação.
- **END_TRANSACTION**: Especifica que as operações **READ** e **WRITE** da transação terminaram, e marca o fim da execução da transação. Entretanto, nesse ponto é necessário verificar se as mudanças introduzidas pela transação podem ser permanentemente aplicadas ao banco de dados (efetivadas), ou se a transação deverá ser abortada porque viola a serialização (Seção 17.5), ou por alguma outra razão.
- **COMMIT_TRANSACTION**: Indica *término com sucesso* da transação, de forma que quaisquer alterações (atualizações) executadas poderão ser seguramente **efetivadas** no banco de dados e não serão desfeitas.
- **ROLLBACK** (ou **ABORT**): Indica que uma transação não *terminou com sucesso*, de forma que quaisquer mudanças ou efeitos que a transação possa ter aplicado ao banco de dados deverão ser desfeitas.

A Figura 17.4 mostra um diagrama de transição de estados que descreve como uma transação percorre seus estados de execução. Uma transação entra em **estado ativo** imediatamente após o início de sua execução, no qual poderá emitir operações **READ** (leitura) e **WRITE** (gravação). Quando a transação termina, ela passa para o **estado de efetivação parcial**. Nesse ponto, alguns protocolos de restauração precisam garantir que uma falha de sistema não impossibilite a gravação permanente das mudanças promovidas pela transação (geralmente pela gravação de mudanças no *log* do sistema, que será visto na próxima seção). Uma vez atendidas todas as verificações, diz-se que a transação alcançou seu ponto de efetivação, e entra, então, no **estado de efetivação (committed state)**. Os pontos para efetivação serão discutidos com mais detalhes na Seção 17.2.3. Uma vez efetivada, a transação tem sua execução concluída com sucesso, e todas as suas mudanças serão gravadas permanentemente no banco de dados.

BEGIN TRANSACTION

ATIVO

FIGURA 17.4 Diagrama de transição de estado ilustrando os estados de execução de uma transação.

Entretanto, uma transação poderá entrar em estado **de falha** se uma das verificações falhar ou se a transação for interrompida durante seu estado ativo. A transação deverá, então, ser revertida para desfazer os efeitos de suas operações **WRITE** no banco de dados. O **estado encerrado** corresponde ao estado da transação quando deixa o sistema. As informações sobre a transação que foram mantidas no sistema em tabelas, enquanto a transação estava em execução, serão removidas quando a transação terminar. Transações falhas ou interrompidas podem ser *reiniciadas* mais tarde — automaticamente ou depois de serem re-submetidas pelo usuário — como uma nova transação.

17.2.2 O Log (Registro de Ocorrências) do Sistema

Para poder se recuperar de falhas que afetam as transações, o sistema mantém um **log** para controlar todas as operações da transação que afetem valores dos itens do banco de dados. Essas informações podem ser necessárias para permitir a restauração de falhas. O *log* é mantido em disco, de forma que não será afetado por nenhum tipo de falha, exceto falha de disco ou catastrófica. Além disso, o *log* é periodicamente copiado para um sistema de armazenamento (fita), a fim de ser protegido de falhas catastróficas.

Relacionaremos, agora, os tipos de entradas — chamadas **registros de log** — gravadas no *log* a cada ação

5 O controle de concorrência otimista (Seção 18.4) também exige que certas verificações sejam feitas nesse ponto para garantir que a transação não interfira em outras transações em execução.

6 O *log*, às vezes, é chamado de diário do SGBD.

executada. Nessas entradas, T faz referência a um único identificador de transação (*id_transação*), que é gerado automaticamente pelo sistema e usado para identificar cada transação:

1. [*start_transaction*, T]: Indica que a transação T começou a ser executada.
2. [*escrever_item*, T, X, *valor_antigo*, *novo_valor*]: Indica que a transação T mudou o valor do item X no banco de dados do *valor_antigo* para o *novo_valor*.
3. [*ler_item*, T, X]: Indica que a transação T leu o valor do item X do banco de dados.
4. [*commit*, T]: Indica que a transação T foi completada com sucesso e afirma que seus efeitos podem ser efetivados (gravados permanentemente) no banco de dados.
5. [*abort*, T]: Indica que a transação T foi interrompida (abortada).

Protocolos para restauração que evitem reversões (*rollbacks* — retornos) em cascata (Seção 17.4.2) — os quais compreendem quase todos os protocolos práticos — não exigem que as operações READ sejam escritas no *log* do sistema. Entretanto, se o *log* também for usado para outros fins — como auditoria (manter o controle de todas as operações do banco de dados) —, então esse tipo de entrada deverá ser incluído. Além disso, alguns protocolos de restauração exigem entradas WRITE mais simples, que não incluem *novo_valor* (Seção 17.4.2).

Observe que supusemos aqui que todas as mudanças permanentes no banco de dados são efeitos das transações, de forma que a noção de restauração de falhas de transação implica um conjunto de ações para desfazer ou refazer, individualmente, operações de transações do *log*. Se o sistema entrar em colapso, poderemos recuperar o banco de dados para um estado consistente examinando o *log* e usando uma das técnicas descritas no Capítulo 19. Como o *log* contém um registro para cada operação WRITE que altera o valor de algum item do banco de dados, é possível desfazer (undo) o efeito dessas operações WRITE de uma transação T, voltando o *log* e reajustando todos os itens alterados por uma operação WRITE de T para seus valores *antigos*. Refazer (redo) as operações de uma transação pode também ser necessário se todas as suas atualizações estiverem registradas no *log*, mas pode ocorrer uma falha antes que se possa garantir que todos esses novos valores foram gravados permanentemente, no banco de dados real do disco. Para refazer as operações da transação T, avança-se no *log* registrando todos os itens alterados por uma operação WRITE de T com seus novos valores.

17.2.3 Ponto de Efetivação (*Commit Point*) de uma Transação

Uma transação T alcança seu ponto de efetivação (*commit point*) quando todas as suas operações que acessam o banco de dados estão sendo executadas com sucesso e o efeito de todas elas estiverem sendo gravados no *log*. Após o ponto de efetivação, a transação é dita efetivada, e seu efeito será gravado de modo *permanente* no banco de dados. Em seguida, a transação grava um registro de efetivação [*commit*, T] no *log*. Se ocorrer uma falha de sistema, buscaremos todas as transações T no *log* que tenham um registro [*start_transaction*, T], mas que ainda não tenham gravado seu registro [*commit*, T]; essas transações deverão se reverter (*rollback*) para que seus efeitos sejam desfeitos no banco de dados durante o processo de restauração. Transações que tenham registro de efetivação (*commit*) no *log* deverão ter também registradas no *log* suas operações WRITE, de forma que seu efeito no banco de dados possa ser *refeito* por meio deles.

Observe que o arquivo de *log* deve ser mantido no disco. Conforme analisado no Capítulo 13, atualizar um arquivo de disco envolve copiar, em um *buffer* da memória principal, o bloco apropriado do arquivo no disco, e depois copiar o *buffer* para o disco. É comum manter um ou mais blocos do arquivo de *log* em *buffers* da memória principal até que eles estejam preenchidos com entradas de *log* e, então, gravá-los de uma vez só, em vez de gravar em disco toda entrada de *log* que for adicionada por vez. Isso reduz carga de sistema (*overhead*) para diversas gravações em disco do mesmo bloco de arquivo de *log*. No momento em que houver colapso do sistema, apenas as entradas de *log* que foram gravadas de volta no disco serão consideradas no processo de restauração, pois o conteúdo da memória principal poderá estar perdido. Portanto, antes que uma transação alcance seu ponto de efetivação, toda porção do *log* que ainda não tiver sido registrada em disco precisará agora ser feita. Esse processo é chamado gravação forçada (*force-writting*) do arquivo de *log* antes da efetivação de uma transação.

17.3 PROPRIEDADES DESEJÁVEIS DAS TRANSAÇÕES

As transações devem possuir algumas propriedades, chamadas propriedades ACID, e elas devem ser impostas pelo controle de concorrência e métodos de restauração do SGBD. As propriedades ACID são as seguintes:

- 7 Desfazer (undo) e refazer (redo) serão discutidos com mais detalhes no Capítulo 19.

17.4 Definindo Plano de Execução (*Schedules*) Baseado na Restaurabilidade (*Recoverability*) 405

1. **Atomicidade:** Uma transação é uma unidade atômica de processamento; ou ela será executada em sua totalidade ou não será de modo nenhum.

2. **Preservação de consistência:** Uma transação será preservadora de consistência se sua execução completa fizer o banco de dados passar de um estado consistente para outro.

3. **Isolamento:** Uma transação deve ser executada como se estivesse isolada das demais. Isto é, a execução de uma transação não deve sofrer interferência de quaisquer outras transações concorrentes.

4. **Durabilidade ou permanência:** As mudanças aplicadas ao banco de dados por uma transação efetivada devem persistir no banco de dados. Essas mudanças não devem ser perdidas em razão de uma falha.

A propriedade da atomicidade exige que executemos uma transação por completo. É responsabilidade do subsistema de restauração de transações do SGBD garantir a atomicidade. Se uma transação falhar por alguma razão, como um colapso de sistema durante sua execução, a técnica de restauração deverá desfazer quaisquer efeitos dessa transação no banco de dados.

A preservação da consistência é geralmente considerada responsabilidade do programador que codifica os programas de banco de dados, ou do módulo do SGBD que garante as restrições de integridade. Recordemos que **um estado do banco de dados** é a coleção de todos os itens de dados armazenados (valores) no banco de dados em um dado momento. Um **estado consistente** do banco de dados satisfaz as restrições especificadas no esquema, bem como quaisquer outras restrições que devam controlar o banco de dados. Um programa de banco de dados deve ser escrito de forma que garanta que, se o banco de dados está em um estado consistente antes da execução da transação, ele estará em um estado consistente depois da execução *completa* da transação, assumindo que *nenhuma interferência com outras transações* ocorrerá.

O isolamento é imposto pelo subsistema de controle de concorrência do SGBD. Se alguma transação não tornar suas atualizações invisíveis às outras até que seja efetivada, será imposta alguma forma de isolamento que solucione o problema de atualizações temporárias e elimine reversões (*rollbacks*) em cascata (Capítulo 19). Tem havido tentativas para estabelecer o *nível de isolamento* de uma transação. Diz-se que uma transação tem nível de isolamento 0 (zero) se ela não sobrescrever leitura de sujeira de transações de nível mais alto. O isolamento de nível 1 (um) não permite atualizações perdidas; e, no isolamento de nível 2, não há atualizações perdidas nem leitura de sujeira. Finalmente, no isolamento de nível 3 (também chamado *isolamento verdadeiro*), há, além das propriedades do nível 2, leituras repetíveis.

Finalmente a propriedade de durabilidade é responsabilidade do subsistema de restauração do SGBD. No Capítulo 19 discutiremos como os protocolos de restauração garantem durabilidade e atomicidade.

17.4 DEFININDO PLANO DE EXECUÇÃO (*SCHEDULES*) BASEADO NA RESTAURABILIDADE (*RECOVERABILITY*)

Quando transações são executadas concorrentemente, de maneira intercalada, a ordem de execução das operações das várias transações é conhecida como **plano de execução** (ou **histórico**). Nesta seção definiremos primeiro o conceito de plano de execução e, depois, caracterizaremos os tipos de planos que facilitam a restauração quando ocorrerem falhas. Na Seção 17.5 caracterizaremos planos em termos de interferência de transações participantes, levando aos conceitos de serialidade e de planos de execução serializáveis.

17.4.1 Planos de Execução (Histórico) de Transações

Um **plano** (ou **histórico**) S de n transações T_1, T_2, \dots, T_n é a ordenação das operações das transações sujeitas a uma restrição tal que, para cada transação T_i que participe de S , as operações de T_i em S deverão aparecer na mesma ordem em que elas ocorrem em T_i . Observe, entretanto, que operações de outras transações T poderão ser intercaladas com as operações de T_i em S . Por ora, considere que a ordem das operações em S seja a *ordenação total*, embora teoricamente seja possível tratar planos cujas operações tenham *ordenações parciais* (conforme veremos depois).

Para fins de restauração e controle de concorrência, estamos interessados principalmente nas operações de *ler_item* (leitura de um item) e *escrever_item* (gravação de um item), bem como nas operações *commit* (efetivação) e *abort* (interrupção). Uma notação resumida para codificação de um plano usa os símbolos r , w , c e a para as operações *ler_item*, *escrever_item*, *commit* e *abort*, respectivamente, e anexa, para documentação, o *id* da transação (número da transação) em cada s .

Veremos os protocolos de controle de concorrência no Capítulo 18.

operação do plano. Nessa notação, o item X do banco de dados que for lido ou gravado segue as operações r e w entre parênteses. Por exemplo, o plano da Figura 17.3a, que chamaremos de S_a , pode ser escrito nesta notação, como segue:

$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$

De forma similar, o plano para a Figura 17.3b, que chamaremos de S_b , pode ser escrito como segue, se assumirmos que a transação T1 abortou depois de sua operação ler_item(Y):

$S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$

Duas operações em um plano são ditas em **conflito** se elas satisfizerem todas as três condições seguintes: (1) pertencerem a diferentes transações; (2) acessarem o mesmo item X; e (3) pelo menos uma das operações ser um escrever_item(X). Por exemplo, no plano S_a , as operações $r_1(X)$ e $w_2(X)$ conflitam, bem como as operações $r_2(X)$ e $w_1(X)$, e as operações $w_1(X)$ e $w_2(X)$. Entretanto, as operações $r_1(X)$ e $r_2(X)$ não conflitam, uma vez que ambas são de leitura; as operações $w_2(X)$ e $w_1(Y)$ não conflitam porque operam em diferentes itens de dados, X e Y; e as operações $r_1(X)$ e $w_1(X)$ não conflitam porque pertencem à mesma transação.

Um plano S, com n transações T1, T2, ..., Tn, é chamado um **plano completo** se forem garantidas as seguintes condições:

1. As operações em S são exatamente as operações de T1, T2, ..., Tn, tendo um *commit* ou um *abort* como última operação de cada transação no plano.
2. Para quaisquer pares de operações da mesma transação Ti, sua ordem de aparecimento em S será a mesma que em Ti.
3. Para quaisquer duas operações conflitantes, uma das duas precisa aparecer antes da outra no plano.

A condição (3) anterior permite que *duas operações não conflitantes* apareçam no plano sem definir qual deva aparecer primeiro, levando, assim, à definição de um plano com ordenação **parcial** das operações das n transações. Entretanto, a ordenação total deve ser especificada em planos com algum par de operações conflitantes (condição 3), e para algum par de operações da mesma transação (condição 2). A condição 1 simplesmente declara que todas as operações das transações devem aparecer em um plano completo. Uma vez que toda transação ou será efetivada ou será abortada, um plano completo não conterá nenhuma transação ativa em seu término.

Em geral, é difícil encontrar planos completos em um sistema de processamento de transações porque novas transações estarão continuamente sendo submetidas ao sistema. Portanto, é útil definir o conceito de projeção de efetivadas (*committed projection*) C(S) de um plano S, que incluirá em S apenas as operações que pertencerem às transações efetivadas — isto é, as transações Ti cuja operação ci *commit*, pertença a S.

17.4.2 Caracterizando um Plano Baseado na Restaurabilidade

Em alguns planos é fácil a restauração de transações, enquanto, em outros, o processo pode ser muito complicado. Portanto, é importante caracterizar os tipos de planos nos quais a restauração é possível, bem como aqueles para os quais a restauração é relativamente simples. Essas caracterizações, na realidade, não fornecem o algoritmo de restauração, apenas tentam caracterizar teoricamente os diferentes tipos de planos.

Primeiro, devemos garantir que, uma vez efetivada a transação T, *nunca* mais seja necessário reverter-la. Os planos que teoricamente buscam esses critérios são chamados de planos *restauráveis* (*recoveráveis*), e aqueles que não buscam são chamados não-restauráveis, portanto, não deveriam ser permitidos. Um plano S é restaurável se nenhuma transação T de S for efetivada até que todas as transações T', que tiverem gravado um item lido por T, tenham sido efetivadas. Uma transação T é **lida** pela transação T' em um plano S se algum item X for gravado por T' e, depois, lido por T. Além disso, T' não deve ser abortada antes que T leia o item X, e não deve haver transações que gravem X depois da gravação de T' e antes da leitura de T (a não ser que alguma dessas transações tiver abortado antes que T tivesse lido X).

Planos restauráveis exigem um processo de restauração complexo, como veremos, mas se forem mantidas informações suficientes (no *log*), poderá ser planejado um algoritmo de restauração. Os planos (parciais) S_a e S_b , da seção anterior, são ambos restauráveis, uma vez que satisfazem a definição acima. Considere o plano S_a' dado abaixo, que é o mesmo plano S_a , exceto pelas duas operações *commit* adicionadas a S_a :

$S_a' = r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$

9 Teoricamente, não é preciso determinar uma ordem entre pares de operações *não-conflitantes*.

10 Na prática, a maioria dos planos tem uma ordenação total de operações. Se for usado processamento paralelo, teoricamente será possível ter planos com operações não-conflitantes, ordenadas parcialmente.

17.5 Definindo Planos de Execução (*Schedules*) Baseados em Serialidade (*Serializability*) 407

S_a é restaurável, ainda que sofra do problema da perda de atualização. Entretanto, considere os dois planos (parciais), S_c e S_d , conforme segue:

S_c : $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2 ; a_1 ;

S_d : $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1 ; c_2 ;

S_e : $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1 ; a_2 ;

S_c não é restaurável porque T_2 lê o item X de T_1 , e T_2 é efetivada antes que T_1 o seja. Se T_1 abortar depois da operação c_2 de S_c , então o valor de X lido por T_2 não será válido, e T_2 precisará ser abortada *após* ter sido efetivada, levando a um plano não restaurável. Para o plano ser restaurável, a operação c_2 em S_c deverá ser adiada até que T_1 seja efetivada, como mostrado em S_d ; se T_1 abortar em vez de efetivar, então T_2 deverá abortar também, como mostrado em S_e , pois o valor de X lido por ela não será mais válido.

Em um plano restaurável, nenhuma transação efetivada jamais deveria precisar ser revertida. Entretanto, é possível ocorrer um fenômeno conhecido como **reversão em cascata** (*cascading rollback* ou **interrupção em cascata** — *cascading abort*), quando uma transação *não-efetivada* tem de ser revertida porque leu um item de uma transação que falhou. Isso é ilustrado no plano S_e , onde a transação T_2 tem de ser revertida porque leu um item X de T_1 , e T_1 foi, então, interrompida.

Como a reversão em cascata pode consumir muito tempo — caso seja preciso reverter muitas transações (Capítulo 19) —, é importante estabelecer planos de execução que garantam que esse fenômeno não ocorra. Um plano é chamado **livre de reversão em cascata** (*avoid cascading rollback* ou *cascadeless* — **livre de cascata**) se cada transação do plano ler somente os itens que foram gravados por transações efetivadas. Nesse caso, todos os itens lidos não serão descartados, logo, não ocorrerá reversão em cascata. Para satisfazer esse critério, o comando $r_2(X)$ dos planos S_d e S_e será adiado até que T_1 seja efetivada (ou abortada), retardando T_2 , mas garantindo que não haja reversão em cascata caso T_1 seja interrompida.

Finalmente, há um terceiro tipo de plano mais restritivo, chamado **plano restrito** (*strict schedule*), no qual as transações não poderão *nem ler nem gravar* um item X até que a última transação que grave X tenha sido efetivada (ou abortada). Planos restritos simplificam o processo de restauração. Em um plano restrito, o processo para desfazer uma operação `escrever_item(X)` de uma transação abortada é simplesmente restaurar a **imagem anterior** (*valor_antigo* ou BFIM — *before image*) do item de dado X. Esse procedimento simples sempre funciona corretamente em planos restritos, mas pode não funcionar em planos restauráveis ou livres de reversão em cascata. Por exemplo, considere o plano S_f

S_f : $w_1(X, 5)$; $w_2(X, 8)$; a_1 ;

Suponha que o valor de X fosse originalmente 9, que é a imagem anterior armazenada no *log* do sistema com a operação $w_1(X, 5)$. Se T_1 abortar, como em S_e , o procedimento de recuperação, que restaura a imagem anterior de uma operação de gravação abortada, irá retornar o valor de X para 9, mesmo que ele já tenha sido alterado para 8 pela transação T_2 , levando, assim, a resultados potencialmente incorretos. Embora o plano S_f seja livre de reversão em cascata, ele não é um plano restrito, uma vez que permite que T_2 grave um item X mesmo que a transação T_2 , que alterou X por último, não tenha ainda sido efetivada (ou interrompida). Um plano restrito não terá esse problema.

Até agora caracterizamos os planos de acordo com os seguintes termos: (1) recuperabilidade, (2) impedimento de reversão em cascata e (3) rigorosidade. Assim, as propriedades dos planos impõem, sucessivamente, condições cada vez mais rigorosas. A condição (2) implica a condição (1), e a condição (3) implica ambas, a (2) e a (1). Dessa forma, todos os planos restritos são livres de reversão em cascata, e todos os planos livres de reversão em cascata são restauráveis.

17.5 DEFININDO PLANOS DE EXECUÇÃO (*SCHEDULES*) BASEADOS EM SERIALIDADE (*SERIALIZABILITY*)

Na seção anterior caracterizamos os planos com base em suas propriedades de recuperabilidade. Caracterizaremos agora os tipos de planos que são considerados corretos quando transações concorrentes estiverem em execução. Suponha que dois usuários — dois empregados que fazem reservas em uma empresa aérea — submetem ao SGBD as transações T_1 e T_2 da Figura 17.2 aproximadamente ao mesmo tempo. Se a intercalação das operações não for permitida, só há dois resultados possíveis:

1. Executar todas as operações da transação T_1 (em sequência), seguidas por todas as operações da transação T_2 (em sequência).
2. Executar todas as operações da transação T_2 (em sequência), seguidas por todas as operações da transação T_1 (em sequência).

408 Capítulo 17 Introdução aos Conceitos e à Teoria do Processamento de Transações

Essas alternativas são mostradas nas figuras 17.5a e b, respectivamente. Se a intercalação de operações for permitida, haverá muitas ordenações possíveis para as operações de cada uma das transações. Dois planos possíveis são mostrados na Figura 17.5c. O conceito de serialidade de planos é usado para identificar quais planos são corretos quando há intercalação das operações das transações na execução dos planos. Esta seção define serialidade e discute como ela pode ser usada na prática.

```
(a)
Tempo
ler_item(X)t
X.X-/V;
escrever_item(X)
leUtem(V)
Y:=Y+N;
escrever_item(V)
ler_item(X);
X:=X+M\
escrever_item(X);
(b)
Tempo
ler_item(X)!
X:=X-N;
escrever_item(X)
ler_item(V)
Y:=Y+N;
escrever_item( V)
ler_item(X);
X:=X+M\
escrever_item(X);
Plano A
Plano B
(c)
Tempo
ler_item(X)( X.X-/V;
escrever_item(X) leMtem(y)
Y:=Y+N; escrever_item(y)
ler_item(X); X=X+M;
escrever_item(X);
ler_item(X)
X:=X-N;
escrever_item(X)
ler_item(V) Y:=Y+N; escrever_item( V)
leMtem(X);
X:=X+M;
escrever_item(X);
Plano C
Plano D
```

FIGURA 17.5 Exemplos de planos serials e não-seriais envolvendo as transações T_1 e T_2 . (a) Plano serial A: T_1 , seguido por T_2 . (b) Plano serial B: T_2 seguido por T_1 . (c) Dois planos não-seriais, C e D, com intercalação de operações.

17.5.1 Planos Serials, Não-Seriais e de Conflitos Serializáveis

Os planos A e B das figuras 17.5a e b são chamados *serials* porque as operações de cada transação são executadas consecutivamente, sem intercalação das operações de outra transação. Em um plano serial, transações inteiras são executadas em ordem serial: T_1 depois T_2 , na Figura 17.5a, e T_2 depois T_1 , na Figura 17.5b. Os planos C e D da Figura 17.5c são chamados *não-seriais* porque cada sequência intercala operações de duas transações.

Formalmente, um plano S é serial se, para cada transação T participante, todas as operações de T forem executadas consecutivamente no plano; de outra forma, o plano será chamado não-serial. Portanto, em um plano serial, apenas uma transação estará ativa por vez — a efetivação (ou interrupção) da transação ativa iniciará a execução da próxima transação. Nenhuma intercalação ocorre em um plano serial. Uma suposição razoável que poderíamos fazer, se considerarmos transações *independentes*, é que cada plano serial seja considerado correto. Isso porque cada transação é considerada correta se executada por si própria (de acordo com a propriedade de preservação da consistência, da Seção 17.3). Portanto, não interessa qual transação seja executada primeiro. Enquanto cada transação for executada do começo ao fim, sem nenhuma interferência de operações de outras transações, conseguiremos um resultado final correto no banco de dados. O problema com planos serials é que eles limitam a concorrência ou a intercalação de operações. Em um plano serial, se uma transação esperar por uma operação de I/O para continuar, não poderemos alternar o processador da CPU para outra transação, logo, um tempo valioso de processamento de CPU será desperdiçado. Além disso, se alguma transação T for longa, as outras operações precisarão esperar que T complete todas as suas operações antes de começarem. Portanto, planos serials são geralmente considerados inaceitáveis na prática.

17.5 Definindo Planos de Execução (*Schedules*) Baseados em Serialidade (*Serializability*) 409

Para ilustrar nossa discussão, considere os planos da Figura 17.5 e assumamos que os valores iniciais dos itens do banco de dados sejam $X = 90$ e $Y = 90$, e que $N = 3$ e $M = 2$. Depois da execução das transações T_1 e T_2 , esperaríamos que os valores do banco de dados fossem $X = 89$ e $Y = 93$, de acordo com o significado das transações. Realmente a execução dos planos seriais A ou B fornecerão resultados corretos. Considere, agora, os planos não-seriais C e D. O plano C (que é o mesmo da Figura 17.3a) resultaria em $X = 92$ e $Y = 93$, com valor errado para X, enquanto o plano D forneceria resultados corretos.

O plano C fornece um resultado errôneo por causa do problema da atualização perdida discutido na Seção 17.1.3; a transação T_2 lê o valor de X antes que seja alterado pela transação T_1 , de forma que apenas o efeito de T_2 em X será refletido no banco de dados. O efeito de T_1 em X será *perdido*, sobrescrito por T_2 , levando ao resultado incorreto do item X. Entretanto, alguns planos não-seriais fornecem resultados corretos, como o plano D. Gostaríamos de determinar quais dos planos não-seriais *sempre* fornecem resultados corretos e quais podem fornecer resultados errôneos. O conceito usado para caracterizar planos dessa maneira é o de serialidade.

Um plano S com n transações é **serializável** se ele for *equivalente a algum plano serial* com as mesmas n transações. Definiremos o conceito de equivalência de planos rapidamente. Observe que há n planos seriais possíveis para n transações, e muitos mais planos não-seriais possíveis. Podemos criar dois grupos separados de planos não-seriais: aqueles que são equivalentes a um (ou mais) dos planos seriais e, portanto, serializáveis; e aqueles que não são equivalentes a *nenhum* plano serial e, portanto, não-serializáveis.

Dizer que um plano S não-serial é serializável é o mesmo que dizer que ele é correto, porque ele é equivalente a um plano serial, que é considerado correto. A questão que permanece é: quando dois planos são considerados 'equivalentes'? Há diversos caminhos para definir equivalência de planos. A definição de equivalência de plano mais simples, mas menos satisfatória, envolve comparar os efeitos dos planos no banco de dados. Dois planos são chamados de resultados **equivalentes** se produzirem o mesmo estado final no banco de dados. Entretanto, dois planos diferentes podem produzir acidentalmente o mesmo estado final. Por exemplo, na Figura 17.6, os planos S_1 e S_2 produzirão o mesmo estado final no banco de dados se forem executados com valor inicial $X = 100$; mas, para outros valores de X, os planos *não* apresentarão resultados equivalentes. Além disso, esses dois planos executam diferentes transações, logo, decididamente, eles não deveriam ser considerados equivalentes. Portanto, a equivalência de resultado sozinha não pode ser usada para definir equivalência de planos. A abordagem mais segura e mais geral para definir equivalência de planos não é fazer qualquer suposição sobre os tipos de operações contidas nas transações. Para dois planos serem equivalentes, as operações aplicadas a cada item de dado afetado por eles deveriam ser aplicadas, em ambos os planos, na *mesma ordem*. São usadas geralmente duas definições de equivalência de planos: *equivalência de conflito* e *equivalência de visão*. A seguir, veremos equivalência de conflito, que é a definição mais comumente usada.

Dois planos são **conflito equivalentes** se a ordem de quaisquer duas operações *conflitantes* for a mesma em ambos os planos.

Lembremos a Seção 17-4.1, que diz que duas operações em um plano estarão em *conflito* se elas pertencerem a diferentes transações, acessarem o mesmo item do banco de dados e pelo menos uma das duas for uma operação escrever_item. Se duas operações conflitantes forem aplicadas em ordens *diferentes* nos dois planos, o efeito final poderá ser diferente no banco de dados ou nas outras transações do plano, portanto, os planos não serão conflitos equivalentes. Por exemplo, se uma operação de leitura e gravação ocorrer na ordem $T_1(X)$, $w_2(X)$ no plano S_1 e na ordem inversa $w_2(X)$, $T_1(X)$ no plano S_2 , o valor lido por $T_1(X)$ pode ser diferente em cada plano. De forma similar, se duas operações de gravação ocorrerem na ordem $w_1(X)$, $w_2(X)$ em S_1 , e a ordem inversa $w_2(X)$, $w_1(X)$ em S_2 , a próxima operação $r(X)$ dos dois planos lerá valores potencialmente diferentes; ou ainda, se elas forem as últimas operações para o item X, seu valor final no banco de dados será diferente.

Usando a noção de equivalência de conflito, definiremos que um plano S é **conflito serializável** se ele for (conflito) equivalente a algum plano serial S'. Nesse caso, poderemos reordenar as operações *não-conflitantes* em S até formar um plano serial equivalente S'. De acordo com essa definição, o plano D da Figura 17.5c será equivalente ao plano serial A da Figura 17.5a. Em ambos os planos, a ler_item(X) de T_2 lê o valor de X escrito por T_1 , enquanto as outras operações ler_item lerão os valores do banco de dados em seu estado inicial. Além disso, T_1 será a última transação a alterar Y, e T_2 será a última transação a alterar X em ambos os planos. Como A é um plano serial e o plano D é equivalente a A, D é um plano *serializável*. Observe que as operações $r_1(Y)$ e $w_1(Y)$ do plano D não conflitam com as operações $r_2(X)$ e $w_2(X)$, uma vez que elas acessam diferentes itens de dado. Assim, podemos colocar $r_1(Y)$, $w_1(Y)$ antes de $r_2(X)$, $w_2(X)$, criando um plano serial equivalente a T_1 , T_2 .

11 Utilizaremos *serializável* com o significado de conflito serializável. Uma outra definição de serializável usada na prática (Seção 17.6) é ter leituras repetíveis, nenhuma leitura de sujeira e nenhum registro fantasma (Seção 18.7.1 para uma discussão sobre fantasmas).

```

S,
ler_item(X);
X:=X+10;
escrever_item(X);
ler_item(X);
X:=X*1.1;
escreveMtem(X);

```

FIGURA 17.6 Dois planos que têm resultados equivalentes quando o valor inicial de $X = 100$, mas que, no geral, não são resultados equivalentes.

O plano C da Figura 17.5c não é equivalente a nenhum dos dois planos seriais possíveis, A e B, portanto, não é *serializável*. Tentar reordenar as operações do plano C para encontrar um plano serial equivalente não é possível, porque $r_2(X)$ e $w_1(X)$ conflitam, o que significa que não poderemos mover $r_2(X)$ para obter um plano serial T_1, T_2 equivalente. Similarmente, como $w_1(X)$ e $w_2(X)$ conflitam, não poderemos mover $w_1(X)$ para obter um plano serial T_2, T_1 equivalente.

Outra definição de equivalência, mais complexa — chamada *equivalência de visão* —, que conduz ao conceito de *serialidade de visão*, será vista na Seção 17.5.4.

17.5.2 Testando o Conflito Serialidade de um Plano

Há um algoritmo simples para determinar o conflito serialidade de um plano. A maioria dos métodos para controle de concorrência *não* testa, de fato, a serialidade. Pelo contrário, muitos protocolos, ou regras, foram desenvolvidos para garantir que um plano seja serializável. Discutiremos aqui o algoritmo para testar o conflito serialidade de planos de execução para compreender melhor os protocolos de controle de concorrência que serão discutidos no Capítulo 18.

O Algoritmo 17.1 pode ser usado para testar um plano para conflito serialidade. O algoritmo apenas verifica as operações `ler_item` e `escrever_item` de um plano e constrói um grafo de precedência (ou grafo de serialização), que é um grafo orientado $G = (N, E)$, que consiste de um conjunto de nós $N = (T_1, T_2, \dots, T_n)$ e de um conjunto de setas dirigidas $E = (e_1, e_2, \dots, e_m)$. Haverá um nó no grafo para cada transação T_i do plano. Cada seta e , no grafo será $(T_j \rightarrow T_k)$, $1 \leq j < n$, $1 \leq k < n$, onde T_j é o nó inicial de e , e T_k é o nó final de e . Uma seta será criada caso alguma das operações de T_j apareça no plano *antes* de alguma *operação conflitante* de T_k .

- (a)
- (b)
- (c)
- (d)

FIGURA 17.7 Construindo os grafos de precedência para teste de conflito serialidade dos planos A a D da Figura 17.5. (a) Grafo de precedência para o plano serial A. (b) Grafo de precedência para o plano serial 6. (c) Grafo de precedência para o plano C (não serializável). (d) Grafo de precedência para o plano D (serializável, equivalente ao plano A).

Algoritmo 17.1: Testando conflito serialidade de um plano S.

1. Para cada transação T_i participante do plano S, criar um nó rotulado T_i no grafo de precedência.
2. Para cada caso em S em que T_i executar um `ler_item(X)` depois que uma T_j executar um `escrever_item(X)`, criar uma seta $(T_j \rightarrow T_i)$ no grafo de precedência.
3. Para cada caso em S em que T_i executar um `escrever_item(X)` depois que T_j executar um `ler_item(X)`, criar uma seta $(T_j \rightarrow T_i)$ no grafo de precedência.
4. Para cada caso em S em que T_i executar um `escrever_item(X)` depois que T_j executar um `escrever_item(X)`, criar uma seta $(T_j \rightarrow T_i)$ no grafo de precedência.
5. O plano S será serializável se, e apenas se, o grafo precedência não contiver ciclos.

17.5 Definindo Planos de Execução (*Schedules*) Baseados em Serialidade (*Serializability*) 411

O grafo de precedência será construído como descrito no Algoritmo 17.1. Se houver um ciclo no grafo de precedência, o plano S não será (conflito) serializável; se não houver ciclo, S será serializável. Um **ciclo** em um grafo orientado é uma **seqüência de setas** $C = ((T^a \rightarrow T_k), (T_k \rightarrow T_p), \dots, (T_l \rightarrow T^a))$, com a propriedade de que o nó inicial de cada seta — exceto a primeira — seja o mesmo nó que o do final da seta anterior, e o nó inicial da primeira seta seja o mesmo que o do final da última seta (a seqüência começa e termina no mesmo nó).

No grafo de precedência, uma seta T_i para T_j significa que a transação T_i precisa preceder a transação T_j : em qualquer plano serial equivalente a S , porque as duas operações conflitantes aparecem no plano nessa ordem. Se não houver ciclo no grafo de precedência, poderemos criar um **plano serial equivalente** S' que é equivalente a S , pela ordenação das transações que pertencem a S , como segue: sempre que houver uma seta no grafo de precedência de T_i para T_j , T_i deverá aparecer antes de T_j no plano serial equivalente S' . Observe que as setas $(T_i \rightarrow T_j)$ em um grafo de precedência podem opcionalmente ser rotuladas pelo(s) nome(s) do(s) item(ns) de dado que motivam sua criação. A Figura 17.7 mostra tais rótulos nas setas.

Em geral, vários planos seriais podem ser equivalentes a S se o grafo de precedência de S não apresentar ciclos. Entretanto, se o grafo de precedência tiver um ciclo, é fácil mostrar que não poderemos criar nenhum plano serial equivalente, assim, S não será serializável. Os grafos de precedência criados para os planos de A a D da Figura 17.5 aparecem, respectivamente, nas figuras 17a a d. O grafo para o plano C tem um ciclo, logo, ele não será serializável. O grafo para o plano D não possui ciclos, assim, ele é serializável, e o plano serial equivalente será T_1 seguido por T_2 . Os grafos para os planos A e B não têm ciclos, como seria esperado, uma vez que os planos são seriais e, portanto, serializáveis.

Um outro exemplo, no qual participam três transações, é mostrado na Figura 17.8. A Figura 17.8a mostra as operações ler_item e escrever_item de cada transação. Dois planos, E e F, para essas transações, são mostrados nas figuras 17.8b e c, respectivamente, e os grafos de precedência para os planos E e F são mostrados nas partes d e e. O plano E não é serializável porque o grafo de precedência correspondente apresenta ciclos. O plano F é serializável, e o plano serial equivalente a F é mostrado na Figura 17.8e. Embora exista apenas um plano serial equivalente para F, em geral, pode haver *mais de um plano serial equivalente* a um plano serializável. A Figura 17.8f mostra um grafo de precedência representando um plano que tem dois planos seriais equivalentes.

17.5.3 Aplicações da Serialidade

Conforme discutimos anteriormente, dizer que um plano S é (conflito) serializável — isto é, S é (conflito) equivalente a um plano serial — equivale a dizer que S está correto. Entretanto, ser *serializável* é diferente de ser *serial*. Um plano serial implica um processamento ineficiente, pois nenhuma intercalação de operações de diferentes transações será permitida. Assim, uma baixa utilização de CPU, enquanto uma transação espera por I/O (entrada/saída) de disco ou pelo término de outra transação, pode levar a um processamento consideravelmente lento. Um plano serializável fornece os benefícios da execução concorrente, sem deixar de ser correto. Na prática, é muito difícil testar a serialidade de um plano. A intercalação de operações de transações concorrentes — que em geral são executadas como processos pelo sistema operacional — é, normalmente, determinada pelo *scheduler* (supervisor) do sistema operacional, que aloca recursos a todos os processos. Fatores como a carga do sistema, o tempo de submissão de transação e as prioridades de processos contribuem para a ordenação das operações de um plano. Assim, é difícil determinar como as operações de um plano serão intercaladas antecipadamente de modo a garantir a serialidade.

Se as transações forem executadas à vontade e, depois, a serialidade do plano resultante for testada, precisaremos cancelar o efeito do plano, caso ele não seja serializável. Esse é um problema sério, que torna essa abordagem impraticável. Assim, a abordagem adotada na maioria dos sistemas práticos é determinar métodos que garantam a serialidade, sem testar os planos propriamente ditos. A abordagem adotada na maioria dos SGBDs comerciais é definir **protocolos** (conjuntos de regras) que — se seguidos por todas as transações individualmente, ou impostos por um subsistema de controle de concorrência do SGBD — garantirão a serialidade de *todos os planos dos quais as transações participem*.

Outro problema aparece aqui: quando as transações são submetidas continuamente ao sistema, é difícil determinar quando um plano começa e quando termina. A teoria da serialidade pode ser adaptada para tratar esse problema, considerando apenas a projeção de um plano S . Recordemos, da Seção 17.4.1, que a *projeção de efetivadas* $C(S)$ de um plano S engloba somente as operações em S que pertencerem às transações efetivadas. Podemos, teoricamente, definir um plano S como serializável se sua projeção de efetivadas $C(S)$ for equivalente a algum plano serial, uma vez que apenas transações efetivadas são garantidas pelo SGBD.

12 Esse processo de ordenação dos nós em um grafo acíclico é conhecido como classificação topológica.

412 Capítulo 17 Introdução aos Conceitos e à Teoria do Processamento de Transações
(a)

ler.
jtem
(X)

escrever
item
W;
ler_
jtem
(Y)

escrever
jtem
(V);

ler
item
(2)

ler_
jtem
(Y)

escrever
item
(V);
ler_
jtem
(X);

escrever
Jtem
W;

ler
item
(VI

ler.
jtem
(Z)

escrever
item
(V);
escrever
Jtem
(2);
(b)

Tempo

transação 7",	transação 7"₂	transação T₃	
	ler_item (2) ler_item (V) escrever_item (V);		ler_item (Y) ler_item (2)
ler_item (X) escrever_item (X);		ler_item (X);	escrever_item (Y) escrever_item (Z)
ler_item (V) escrever_item (V);		escrever_item (X);	

(b)

Tempo

transação 7"i	transação T₂	transação 7₃
		ler_item (Y) ler_item (2)
lerItem (X) escreverItem (X);		ler_item (2) escrever_item (Y) escrever_item (2)
ler_item (Y) escrever_item (Y);		ler_item (V) escrever_item (V); ler_item (X); escrever_item (X);

FIGURA 17.8 Outro exemplo de teste de serialidade, (a) As operações READ e WRITE das três transações 7., T₂ e T₃. (b) Plano F (c) Plano F.

(d)

Planos seriais equivalentes

Nenhuma Razão

ciclo $X(T_1^>T_2), Y(T_2^>T_3)$ ciclo $x(T_1, -T_2), yz(T_2 - T_3), v(T_3 - T_1)$

(e)

Planos seriais equivalentes

(f)

Planos seriais equivalentes

 T_3 T_3 T_3 T_2 T_2

FIGURA 17.8 Outro exemplo de teste de serialidade. (d) Grafo de precedência para o plano *E*. (e) Grafo de precedência para o plano *F*. (f) Grafo de precedência com dois planos seriais equivalentes, (conf/nuação)

No Capítulo 18 veremos outros protocolos de controle de concorrência que garantem a serialidade. A técnica mais comum, chamada *bloqueio em duas fases* (*two-phase locking*), é baseada no bloqueio de itens de dados de modo a impedir que transações concorrentes interfiram umas nas outras e para impor uma condição adicional que garanta a serialidade. Isso é usado na maioria dos SGBDs comerciais. Outros protocolos têm sido propostos, entre eles, o de *ordenação por marca de tempo* (*timestamp ordering*), no qual cada transação recebe uma única marca de tempo (*timestamp*) e o protocolo garante que toda operação conflitante seja executada na sequência dos *timestamps* da transação; *protocolos de multiversões*, baseados na manutenção de diversas versões dos itens de dados; e *protocolos otimistas* (também chamados de *certificação* ou *validação*), que verificam possíveis violações de serialidade depois que as transações terminam, mas que, antes, permitem que sejam efetivadas.

17.5.4 Equivalência de Visão e Visão Serialidade

Na Seção 17.5.1 definimos os conceitos de equivalência de conflito entre planos e conflito serialidade. Uma outra definição, menos restritiva, de equivalência de planos, é chamada de *equivalência de visão*. Ela leva a uma outra definição de serialidade chamada *visão serialidade*. Diz-se que dois planos *S* e *S'* são visão equivalentes se as três condições seguintes forem atendidas:

1. O conjunto de transações participantes em *S* e *S'* seja o mesmo e que *S* e *S'* contenham as mesmas operações dessas transações.

13 Estes protocolos não têm sido usados com muita frequência; a maioria dos sistemas utiliza uma variação de protocolo de bloqueio em duas fases.

414 Capítulo 17 Introdução aos Conceitos e à Teoria do Processamento de Transações

2. Para toda operação $r_i(X)$ de T_i em S , se o valor X lido pela operação tiver sido alterado por uma operação $w_j(X)$ i (ou se ele for o valor original de X antes do plano iniciar), a mesma condição deverá ser garantida para o valor X pela operação $r_i(X)$ de T_i em S' .
3. Se a operação $w_k(Y)$ de T_k for a última operação a gravar o item Y em S , então $w_j(Y)$ de T_k também deverá ser a última operação a gravar Y em S' .

A idéia básica da equivalência de visão é a de que, enquanto cada operação de leitura de uma transação estiver lendo o resultado da mesma operação de gravação em ambos os planos, as operações de gravação de cada transação precisam processar os mesmos resultados. Assim, diz-se que as operações de leitura *têm a mesma visão* em ambos os planos. A condição 3 garante que a operação final de gravação de cada item de dado será a mesma em ambos os planos, assim, o estado final do banco de dados será o mesmo em qualquer um dos planos. Diz-se que o plano S tem visão serializável se ele tiver visão equivalente ; plano serial.

As definições de conflito serialidade e visão serialidade serão similares se uma condição, conhecida como **hipótese de gravação restrita**, for assegurada para todas as transações dentro do plano. Essa condição estabelece que qualquer operação de gravação $w_i(X)$ em T_i será precedida por uma $r_i(X)$ em T_i , e que o valor gravado por uma $w(X)$ em T_i dependerá apenas de um valor de X lido por $r_i(X)$. Ela supõe que o cálculo do novo valor de X é uma função $f(X)$ baseada no valor antigo de X lido no bfe de dados.

Entretanto, a definição de visão serialidade é menos restritiva que a de conflito serialidade sob a **hipótese de grav; irrestrita**, onde o valor gravado por uma operação $w^A(X)$ de T_i pode ser independente de seu valor antigo. Esta é chamada gravação cega e será ilustrada pelo seguinte plano S com três transações T_1 : $T_1(X)$ $w_1(X)$; T_2 : $w_2(X)$ e T_3 : $w_3(X)$:

S_g : $r_1(X)$; $w_2(X)$; $w_3(X)$; $UijfXj$,-q; c_2 ; c_3 .

Em S_g , as operações $w_2(X)$ e $w^A(X)$ são gravações cegas, uma vez que T_2 e T_3 não lêem o valor de X . O plano S é v serializável, uma vez que é visão equivalente ao plano serial T_1 , T_2 , T_3 . Entretanto, S_g não é conflito serializável, uma vez não é equivalente a nenhum plano serial. Tem sido demonstrado que qualquer plano conflito serializável também é visã rializável, mas não vice-versa, como ilustrado pelo exemplo anterior. Há um algoritmo para testar se um plano S é visão s« lizável ou não. Entretanto, o problema do teste de visão serialidade tem se mostrado NP-hard, o que significa ser altam< improvável encontrar um algoritmo de tempo polinomial eficiente para esse problema.

17.5.5 Outros Tipos de Equivalência de Planos

A serialização de planos é, às vezes, considerada muito restritiva como condição para garantir a correteude de execuções c correntes. Algumas aplicações podem produzir planos corretos satisfazendo condições menos rigorosas que a de conflito ser dade ou visão serialidade. Um exemplo é o tipo conhecido como transação débito-crédito — por exemplo, aquelas que a cam depósitos e retiradas de um item de dado cujo valor é o saldo corrente de uma conta bancária. A semântica da opera de débito-crédito é atualizar o valor de um item de dado X pela subtração ou pela adição de um valor. Visto que as operação adição e subtração são comutativas — isto é, elas podem ser aplicadas em qualquer ordem —, é possível produzir planos i retos, que não sejam serializáveis. Por exemplo, considere as duas transações a seguir, cada uma das quais poderia ser us para transferir uma quantia de dinheiro entre duas contas bancárias:

T_1 : $r_1(X)$; $X := X - 10$; $w_1(X)$; $r_1(Y)$; $Y := Y + 10$; $w_1(Y)$; T_2 : $r_2(Y)$; $Y := Y - 20$; $w_2(Y)$; $r_2(X)$; $X := X + 20$; $w_2(X)$;

Considere o seguinte plano não-serializável S_h das duas transações:

S_h : $r_1(X)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$;

Com o conhecimento adicional, ou **semântico**, de que as operações entre cada $r_i(l)$ e $tf_i(7)$ são comutativas, sabe-se que a ordem de execução das seqüências (ler, atualizar, gravar) não é tão importante quanto não interromper cada seqüer (ler, atualizar, gravar) de uma dada transação T_i para um item í em particular por causa de operações conflitantes. Portanto plano S_j , é considerado correto ainda que não seja serializável. Pesquisadores têm trabalhado para ampliar a teoria do conti de concorrência de modo a tratar os casos em que a capacidade de serialização é considerada muito restritiva como condii para exatidão de planos.

1 7.6 Suporte de Transações em SQL

415

17.6 SUPORTE DE TRANSAÇÕES EM SQL

A definição de uma transação SQL é similar aos conceitos de transação já definidos. Isto é, ela é uma unidade lógica de trabalho e é garantida como atômica. Uma única declaração SQL é sempre considerada atômica — ou sua execução é completa e sem erros ou ela falha e deixa o banco de dados inalterado.

Em SQL não há declaração `BEGIN TRANSACTION` explícita. O início da transação é implícito e ocorre quando declarações SQL são encontradas. Entretanto, toda transação precisa ter uma declaração explícita de término, um `COMMIT` ou um `ROLLBACK`. Toda transação tem determinadas características que são especificadas pela declaração `SET TRANSACTION` em SQL. As características são o *modo de acesso*, o *tamanho da área de diagnóstico* e o *nível de isolamento*.

O **modo de acesso** pode ser especificado por `READONLY` (somente leitura) ou `READ WRITE` (leitura e gravação). O padrão é `READ WRITE`, a não ser que seja especificado `READ UNCOMMITTED` (leitura não efetivada — veja a seguir) como nível de isolamento, quando será assumido `READ ONLY`. O modo `READ WRITE` permite atualização, inserção, remoção e criação de comandos a serem executados. O modo `READ ONLY`, como o nome indica, é simplesmente para recuperação de dados.

A opção para o **tamanho da área de diagnóstico**, `DIAGNOSTIC SIZE n`, especifica um valor inteiro *n*, indicando o número de condições que podem ser manipuladas simultaneamente na área de diagnóstico. Essas condições fornecem informações sobre as condições de execução (erros ou exceções), ao usuário ou ao programa, para a maior parte das declarações SQL executadas mais recentemente.

A opção **nível de isolamento** é especificada pela declaração `ISOLATION LEVEL <nível de isolamento>`, onde o valor de *<nível de isolamento>* pode ser `READ UNCOMMITTED` (leitura não efetivada), `READ COMMITTED` (leitura efetivada), `REPEATABLE READ` (leitura repetível) ou `SERIALIZABLE` (serializável). O nível de isolamento padrão é `SERIALIZABLE`, entretanto, alguns sistemas usam como padrão o `READ COMMITTED`. O uso do termo `SERIALIZABLE` tem por base a não permissão de violações que causem leitura de sujeira, leitura não repetível e fantasmas, assim, ele não é idêntico à forma de serialização definida anteriormente na Seção 17.5. Se uma transação for executada em um nível de isolamento mais baixo que o `SERIALIZABLE`, podem ocorrer uma ou mais das três seguintes violações:

1. **Leitura de sujeira:** Uma transação T_1 pode ler uma atualização ainda não efetivada de uma transação T_2 . Se T_2 falhar e for abortada, então T_1 lera um valor que não existe e está incorreto.

2. **Leitura não-repetível:** Uma transação T_1 pode ler um dado valor em uma tabela. Se, depois, uma outra transação T_2 atualizar esse valor e T_1 lê-lo novamente, T_1 enxergará um valor diferente.

3. **Fantasmas:** Uma transação T_1 pode ler um conjunto de linhas de uma tabela, provavelmente baseada em alguma condição especificada na cláusula `WHERE` SQL. Suponha, agora, que uma transação T_2 insira uma nova linha que também satisfaça a condição da cláusula `WHERE` usada em T_1 dentro da tabela usada por T_1 . Se T_1 for repetida, então verá um fantasma, uma linha que não existia anteriormente.

A Tabela 17.1 resume as possíveis violações nos diferentes níveis de isolamento. Uma entrada 'sim' indica que a violação é possível e uma entrada 'não' indica que não é.

Possíveis Violações Baseadas em Isolamento Níveis como Definidos em SQL

Nível de isolamento

Tipo de Violação

Leitura suja

Não-repetível

Fantasma

LEITURA NÃO EFETIVADA LEITURA EFETIVADA LEITURA REPETITIVA SERIALIZÁVEL

sim não não não

sim sim não não

sim sim sim não

Uma transação SQL simples poderia ser da seguinte forma:

`EXEC SQL WHENEVER SQLERROR GOTO UNDO; EXEC SQL SET TRANSACTION`

14 Essas são similares aos níveis de isolamento, discutidos brevemente ao final da Seção 17.3.

15 O problema de leitura de sujeira e de leitura não repetível é apresentado na Seção 17.1.3. Os fantasmas são apresentados na Seção 18.6.1.

READ WRITE

DIAGNOSTIC SIZE 5

ISOLATION LEVEL SERIALIZABLE; EXEC SQL INSERT INTO EMPREGADO (PNOME, UNOME, SSN, DNO, SALÁRIO)

VALUES ('Robert', 'Smith', '991004321', 2, 35000); EXEC SQL UPDATE EMPREGADO

SET SALÁRIO = SALÁRIO * 1.1 WHERE DNO = 2; EXEC SQL COMMIT; GOTO THE_END; UNDO: EXEC SQL ROLLBACK; THE_END;

....;

Primeiro, a transação acima insere uma nova linha na tabela EMPREGADO; depois, atualiza o salário de todos os empregados que trabalham no departamento 2. Se ocorrer um erro em alguma declaração SQL, a transação inteira será revertida. Com isso, quaisquer alterações de salário (provenientes dessa transação) precisariam ser restauradas aos seus antigos valores e a linha inserida seria removida.

Como podemos ver, a SQL fornece várias facilidades para o tratamento de transações. O DBA ou os programadores do banco de dados podem tirar vantagem dessas opções tentando melhorar o desempenho de transações pelo relaxamento da serialização, caso seja aceitável em suas aplicações.

17.7 RESUMO

Neste capítulo discutimos os conceitos de processamento de transação. Introduzimos o conceito de uma transação de banco de dados e as operações relevantes ao processamento de transações. Comparamos sistemas monousuários com sistemas multiusuários e apresentamos exemplos de como execuções descontroladas de transações concorrentes em um sistema multiusuário podem levar a resultados e valores incorretos do banco de dados. Vimos também os vários tipos de falhas que podem ocorrer durante a execução de uma transação.

Em seguida, apresentamos os estados típicos pelos quais uma transação passa durante sua execução e discutimos diversos conceitos que são usados para restauração e métodos de controle de concorrência. O *log* do sistema registra os acessos ao banco de dados, e o sistema usa essas informações para restauração em caso de falhas. A transação ou será bem-sucedida, alcançando seu ponto de efetivação, ou falhará e será revertida. Uma transação efetivada terá suas alterações permanentemente gravadas no banco de dados. Apresentamos uma visão das propriedades desejáveis das transações — a saber, atomicidade, preservação de consistência, isolamento e durabilidade — que são, frequentemente, referidas como propriedades ACID.

Definimos, depois, um plano (ou histórico) como uma sequência de operações de diversas transações, com possíveis intercalações. Caracterizamos os planos em termos de sua restaurabilidade. Planos restauráveis garantem que, uma vez efetuada uma transação, ela nunca precise ser desfeita. Planos livres de cascatas agregam uma condição extra para garantir que nenhuma transação interrompida exija a interrupção em cascata de outras transações. Planos restritos fornecem uma condição ainda mais forte, que permite um esquema simples de restauração, consistindo no retorno dos antigos valores dos itens que tenham sido alterados por uma transação abortada.

Definimos, então, equivalência de planos, e vimos que um plano serializável é equivalente a algum plano serial. Definimos os conceitos de equivalência de conflito e equivalência de visão, que levam à definição de conflito serializável e de visão serializável. Um plano serializável é considerado correto. Apresentamos algoritmos para teste de serializabilidade (conflito) de um plano. Discutimos por que o teste de serializabilidade é impraticável em sistemas reais, embora ele possa ser usado para definir e verificar protocolos de controle de concorrência, e mencionamos definições menos restritivas de equivalência de planos. Finalmente, demos uma breve visão de como os conceitos de transação são usados na prática, com SQL.

Discutiremos os protocolos de controle de concorrência no Capítulo 18 e os protocolos para restauração no Capítulo 19.

Questões de Revisão

- 17.1. O que é execução concorrente de transações em banco de dados num sistema multiusuário? Diga por que o controle de concorrência é necessário e dê exemplos informais.
- 17.2. Discuta os diferentes tipos de falhas. O que é falha catastrófica?
- 17.3. Discuta as ações tomadas pelas operações `ler_item` e `escrever_item` em um banco de dados.
- 17.4. Trace um diagrama de estado e analise os estados pelos quais uma transação passa durante sua execução.

17.7 Resumo

417

17.5. Para que o *log* do sistema é usado? Quais são os tipos de registros em um *log*? O que são pontos de efetivação de transação e por que são importantes?

17.6. Discuta as propriedades de atomicidade, durabilidade, isolamento e preservação de consistência de uma transação de banco de dados.

17.7. O que é um plano (histórico)? Defina os conceitos de planos restauráveis, livre de cascata e restritos; compare-os em termos de sua restaurabilidade.

17.8. Discuta as diferentes medidas de equivalência de transação. Qual é a diferença entre equivalência de conflito e equivalência de visão?

17.9. O que é um plano serial? O que é um plano serializável? Por que um plano serial é considerado correto? Por que um plano serializável é considerado correto?

17.10. Qual é a diferença entre as hipóteses de gravação restrita e as de gravação irrestrita? Qual é a mais realista?

17.11. Discuta como a serializabilidade é usada para garantir o controle de concorrência em um sistema de banco de dados. Por que ela é, às vezes, considerada muito restritiva como medida de exatidão de planos?

17.12. Descreva os quatro níveis de isolamento em SQL.

17.13. Defina as violações causadas por leitura de sujeira, leitura não-repetível e fantasmas.

Exercícios

17.14. Mude a transação T_2 , na Figura 17.2b, para ler $\text{ler_item}(X, >0)$;

$X := X + A_i$;

If $X > 90$ then exit

ei se escrever_item(X);

Discuta o resultado final dos diferentes planos das Figuras 17.3a e b, onde $M = 2$ e $N = 2$, com respeito às seguintes questões. A adição da condição acima muda o resultado final? O resultado obedece à regra de consistência associada (que a capacidade de X é 90)?

17.15. Repita o Exercício 17.14 adicionando uma verificação em T_1 , de forma que Y não exceda 90.

17.16. Adicione uma operação *commit* no final de cada uma das transações T_1 e T_2 da Figura 17.2; depois, liste todos os planos possíveis para as transações modificadas. Determine quais planos são restauráveis, quais são livres de cascata e quais são restritos.

17.17. Liste todos os planos possíveis para as transações T_1 e T_2 da Figura 17.2 e determine quais são conflito serializáveis (correto) e quais não são.

17.18. Quantos planos *seriais* existem para as três transações da Figura 17.8a? Quais são? Qual é o número total de planos possíveis?

17.19. Escreva um programa para criar todos os planos possíveis para as três transações da Figura 17.8a, e para determinar quais deles são conflito serializáveis e quais não são. Para cada plano conflito serializável, seu programa deve imprimir o plano e listar todos os planos seriais equivalentes.

17.20. Por que, em SQL, é necessária uma declaração explícita para término de transação, embora não seja necessária uma declaração explícita para início?

17.21. Descreva situações nas quais cada um dos diferentes níveis de isolamento seria usado para o processamento de transações.

17.22. Quais dos seguintes planos é serializável (conflito)? Para cada plano serializável, determine os planos seriais equivalentes.

a. $r_1(X); r_3(X); w_1(X); r_2(X); w_2(X)$;

b. $r_1(X); r_3(X); w_1(X); w_2(X); r_2(X)$;

c. $r_3(X); r_2(X); w_1(X); r_1(X); w_2(X)$;

d. $r_3(X); r_2(X); r_1(X); w_2(X); v_{>X}(X)$.

17.23. Considere as três transações, T_1 , T_2 e T_3 , e os planos S_1 e S_2 abaixo. Trace os grafos (de precedência) de serialidade para S_1 e S_2 diga se são serializáveis ou não. Se um plano for serializável, escreva o(s) plano(s) serial(is) equivalen-te(s).

$T_1: r_1(Z); r_1(Y); w_1(Z); w_1(Y)$;

$T_2: r_2(Z); r_2(Y); w_2(Z); w_2(Y)$;

$T_3: r_3(X); r_3(Y); \llcorner_3(Y)j$

$S_1: r_1(X); r_2(Z); r_1(Z); r_3(X); w_1(X); w_3(Y); r_2(Y); u_{>2}(Z); w_2(Y);$

$S_2: r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); u_{/2}(Z); w_3(Y); u^{\wedge}Y;$

17.24. Considere os planos S_3 , S_4 e S_5 abaixo. Determine se são restritos, livres de cascatas, restauráveis ou não (determine condição de restauração mais estrita que cada plano satisfaz).

$S_3: r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); c_1; w_3(Y); r_2(Y); w_2(Z); u_{/2}W; c_2; S_4: r_1(X); r_2(Z); r_1(Z); r_3(X); r^{\wedge}ju/^{\wedge}X; w_3(Y); r_2(Y);$

$u_{/2}(Z); w_2(Y); c_1; c_2; c_3; S_5: r_1^{\circ}(Z); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); ^{\wedge}(X); c_1; w_2(Z); w_3(Y); u_{>2}(Y); c_3; c_2;$

Bibliografia Seleccionada

O conceito de transação é discutido em Gray (1981). Bernstein, Hadzilacos e Goodman (1987) enfocam as técnicas de controle de concorrência e restauração tanto em sistemas de banco de dados centralizados quanto nos distribuídos; é uma excelente referência. Papadimitriou (1986) oferece uma perspectiva mais teórica. Gray e Reuter (1993) oferecem, em um grande livro de referência, com mais de mil páginas, uma perspectiva mais prática dos conceitos e das técnicas do processamento de transações. Elmagarmid (1992) e Bhargava (1989) fornecem um conjunto de artigos sobre pesquisas em processamento de transações. O suporte a transações em SQL é descrito em Date e Darwen (1993). Os conceitos de serialidade são introduzidos por Gray *et al.* (1975). Visão de serialidade é definida em Yannakakis (1984). Restaurabilidade de planos é discutida em Hadzilac (1983, 1988).

Neste capítulo discutiremos técnicas de controle de concorrência usadas para assegurar a propriedade da não interferência ou isolamento de transações que são executadas concorrentemente. A maioria dessas técnicas assegura a serialização de planos de execução (Seção 17.5) usando **protocolos** (isto é, conjunto de regras) que garantem a serialização. Um importante conjunto de protocolos emprega a técnica de **bloqueio** dos itens de dados para impedir que múltiplas transações acessem os itens concorrentemente; alguns desses protocolos de bloqueio são descritos na Seção 18.1. Protocolos de bloqueio são usados na maioria dos SGBDs comerciais. Um outro conjunto de protocolos de controle de concorrência usa **timestamps (marcas de tempo)**. Um *timestamp* é um identificador único para cada transação, gerado pelo sistema. Protocolos de controle de concorrência que usam ordenação por *timestamp* para assegurar serialização serão descritos na Seção 18.2. Veremos na Seção 18.3 protocolos de controle de concorrência de **multiversão**, que usam múltiplas versões de um item de dado. Na Seção 18.4, apresentaremos um protocolo baseado no conceito de **validação** ou **certificação** de uma transação depois que ela executa suas operações; estes são, às vezes, chamados **protocolos otimistas**.

Um outro fator que afeta o controle de concorrência é a **granularidade** dos itens de dados — isto é, qual porção do banco de dados um item de dado representa. Um item pode ser tão pequeno quanto o valor de um único atributo (campo) ou tão grande quanto um bloco de disco, ou todo um arquivo, ou um banco de dados inteiro. Analisaremos granularidade de itens na Seção 18.5.

Discutiremos, na Seção 18.6, resultados de controle de concorrência que aparecem quando índices são usados para processar transações. Finalmente, na Seção 18.7, veremos alguns resultados adicionais de controle de concorrência.

Se o interesse maior do leitor for a introdução das técnicas de controle de concorrência que são usadas mais freqüentemente na prática, será suficiente ler as seções 18.1, 18.5, 18.6 e 18.7 e, possivelmente, a 18.3.2. As outras técnicas são principalmente de interesse teórico.

18.1 TÉCNICAS DE BLOQUEIO EM DUAS FASES PARA CONTROLE DE CONCORRÊNCIA

Algumas das principais técnicas usadas para controle de execução concorrente de transações são baseadas no conceito de bloqueio de itens de dados. Um **bloqueio (lock)** é uma variável associada a um item de dados que descreve a condição do item em relação às possíveis operações que podem ser aplicadas a ele. Geralmente, há um bloqueio para cada item de dado no banco de dados e eles são usados como meio de sincronizar o acesso por transações concorrentes aos itens do banco de dados. Na Seção 18.1.1 discutiremos a natureza e os tipos de bloqueios. E, na Seção 18.1.2, apresentaremos os protocolos que usam bloqueio para garantir serialização de planos de execução de transações. Finalmente, na Seção 18.1.3, veremos dois problemas associados ao uso de bloqueios — isto é, deadlock (impasse) e starvation (inanição) —, e mostraremos como esses problemas são controlados.

18.1.1 Tipos de Bloqueios e Tabelas de Bloqueio de Sistema

Diversos tipos de bloqueios são usados no controle de concorrência. Para introduzir conceitos de bloqueio gradualmente, veremos primeiro os bloqueios binários, que são simples, mas muito restritivos, portanto, não são usados na prática. Analisaremos, depois, bloqueios compartilhados/exclusivos, que fornecem mais capacidades gerais de bloqueio e são usados em esquemas práticos de bloqueio de banco de dados. Na Seção 18.3.2 descreveremos um bloqueio de certificação e mostraremos como ele pode ser usado para melhorar a execução dos protocolos de bloqueio.

Bloqueios Binários. Um bloqueio binário pode ter dois estados ou valores: bloqueios e desbloqueios (ou 1 e 0, para simplificar). Um bloqueio distinto é associado a cada item X do banco de dados. Se o valor do bloqueio em X for 1, o item X *não pode ser acessado* por uma operação de banco de dados que solicite o item. Se o valor do bloqueio em X for 0, o item pode ser acessado quando solicitado. Referimo-nos ao valor corrente (ou estado) do bloqueio associado a um item X como LOCK(X).

Duas operações, lock_item e unlock_item, são usadas com bloqueio binário. Uma transação solicita acesso a um item X pelo primeiro resultado de uma operação lock_item(X). Se LOCK(X) = 1, a transação é forçada a esperar. Se LOCK(X) = 0, ela aponta para 1 (a transação bloqueia o item) e a transação é permitida para acessar o item X. Quando a transação está usando o item completamente, ela resulta em uma operação unlock_item(X), que aponta LOCK(X) para 0 (desbloqueia o item), de forma que X possa ser acessado por outras transações. Portanto, um bloqueio binário assegura exclusão mútua no item de dado. Uma descrição das operações lock_item(X) e unlock_item(X) é mostrada na Figura 18.1.

Observe que as operações lock_item e unlock_item devem ser implementadas como unidades indivisíveis (conhecidas como seções críticas em sistemas operacionais); isto é, nenhuma intercalação deveria ser permitida, uma vez que uma operação lock ou unlock é iniciada, até que a operação termine ou a transação esteja em estado de espera (wait). Na Figura 18.1, o comando wait com a operação lock_item(X) geralmente é implementado colocando a transação em uma fila de espera pelo item X, até X ser desbloqueado e a transação puder conceder acesso a ele. Outras transações que também querem acessar X são colocadas na mesma fila. Portanto, o comando wait é considerado fora da operação lock_item.

lock item(-Y)

```
B: if LOCK(/)=0 (*item está desbloqueado*) then LOCK (X)<-1 (*bloquear o item*)
else begin wait (até que lockW = 0 e
o gerenciador de bloqueio reinicia a transação); go to B end;
```

unlock item (X):

```
LOCK 0<-0; (*desbloquear o item*) se nenhuma transação estiver esperando então reiniciar uma das transações em espera;
```

FIGURA 18.1 Operações de bloqueio e desbloqueio para bloqueios binários.

Observe que é muito simples implementar um bloqueio binário; tudo o que é necessário é uma variável binário-valorada, LOCK, associada a cada item de dado X no banco de dados. Em sua forma mais simples, cada bloqueio pode ser um registro com três campos: <nome do item de dado, LOCK, transação de bloqueio>, mais uma fila para as transações que estão esperando para acessar o item. O sistema precisa manter apenas esses registros para os itens que estão bloqueados em uma tabela de bloqueio, que pode ser organizada como um arquivo *hash*. Os itens que não estão na tabela de bloqueio são considerados desbloqueados. O SGBD tem um subsistema gerenciador de bloqueio para manter e controlar o acesso aos bloqueios.

Se o esquema simples de bloqueio binário que descrevemos for usado, toda transação deve obedecer às seguintes regras:

1. Uma transação T deve garantir a operação lock_item(X) antes que qualquer operação ler_item(X) ou escrever_item(X) seja executada em T.
2. Uma transação T deve garantir a operação unlock_item(X) depois que todas as operações ler_item(X) e escrever_item(X) sejam completadas em T.
3. Uma transação T não resultará em uma operação lock_item(X) se ela já tiver o bloqueio no item X.

1 Esta regra pode ser desconsiderada se modificarmos a operação lock_item(X) na Figura 18.1 de tal forma que, se o item for correntemente bloqueado pela transação *solicitante*, o bloqueio estará assegurado.

18.1 Técnicas de Bloqueio em Duas Fases para Controle de Concorrência 421

4. Uma transação T não resultará em uma operação `unlock_item(X)`, a menos que ela já tenha o bloqueio no item X. Essas regras podem ser impostas pelo módulo gerenciador de bloqueio do SGBD. Entre as operações `lock_item(X)` e `unlock_item(X)` na transação T, diz-se que T controla o bloqueio no item X. No máximo, uma transação pode controlar o bloqueio de um dado item, assim, duas transações nunca podem acessar o mesmo item ao mesmo tempo.

Bloqueios Compartilhados/Exclusivos (ou Leitura/Escrita). O esquema anterior de bloqueio binário é muito restritivo para itens de banco de dados porque, no máximo, uma transação pode assumir o bloqueio em um dado item. Deveríamos permitir a diversas transações acessarem o mesmo item X, se todas elas acessassem X *apenas com propósito de leitura*. Entretanto, se uma transação for alterar um item X, ela deve ter acesso exclusivo a X. Para esse propósito, um tipo diferente de bloqueio, chamado bloqueio de múltiplo-modo, é usado. Nesse esquema — chamado bloqueios compartilhados/ exclusivos ou de leitura/escrita — há três operações de bloqueio: `read_lock(X)`, `write_lock(X)` e `unlock(X)`. Um bloqueio associado a um item X, `LOCK(X)`, tem, agora, três possíveis estados: 'read_locked', 'write_locked' ou 'unlocked'. Um item read-locked também é chamado de bloqueado-compartilhado (share-locked), porque permite que outras transações leiam o item, enquanto um item write-locked é chamado de bloqueado-exclusivo (exclusive-locked), porque uma transação única controla exclusivamente o bloqueio no item.

Um método para implementação das três operações anteriores em um bloqueio leitura/escrita é manter o controle do número de transações que controlam o bloqueio compartilhado (leitura) em um item, em uma tabela de bloqueio. Cada registro na tabela de bloqueio terá quatro campos: <nome do item de dado, LOCK, num_de_leituras, transacao(oes)_bloqueio>. Novamente, para economizar espaço, o sistema precisa manter registros de bloqueio apenas para os itens bloqueados na tabela de bloqueio. O valor (estado) de LOCK ou é bloqueado para leitura ou escrita, adequadamente codificado (se assumirmos que nenhum registro é mantido na tabela de bloqueio para os itens desbloqueados). Se `LOCK(X) = write-locked`, o valor de `transacao(oes)_bloqueio` é uma única transação que controla o bloqueio exclusivo (escrita) em X. Se `LOCK(X) = read-locked`, o valor de `transacao(oes)_bloqueio` é uma lista de uma ou mais transações que controlam o bloqueio compartilhado (leitura) em X. As três operações `read_lock(X)`, `write_lock(X)` e `unlock(X)` são descritas na Figura 18.2. Conforme descrito anteriormente, cada uma das três operações deve ser considerada indivisível; nenhuma intercalação deve ser permitida, uma vez que uma operação é iniciada até que ou quando a operação termine assumindo o bloqueio, ou, então, seja colocada em uma fila de espera para o item.

Quando usamos o esquema de bloqueio compartilhado/exclusivo, o sistema deve impor as seguintes regras:

1. Uma transação T deve garantir a operação `read_lock(X)` ou `write_lock(X)` antes de qualquer operação `ler_item(X)` ser executada em T.
2. Uma transação T deve garantir a operação `write_lock(X)` antes de qualquer operação `escrever_item(X)` ser executada em T.
3. Uma transação T deve garantir a operação `unlock(X)` depois que todas as operações `ler_item(X)` e `escrever_item(X)` são completadas em T.
4. Uma transação T não vai gerar uma operação `read_lock(X)` se ela já controlar um bloqueio de leitura (compartilhado) ou um bloqueio de escrita (exclusivo) no item X. Essa regra pode ser relaxada, conforme discutiremos brevemente.
5. Uma transação T não resultará uma operação `write_lock(X)` se ela já controlar um bloqueio de leitura (compartilhado) ou um bloqueio de escrita (exclusivo) no item X. Essa regra pode ser relaxada, conforme discutiremos brevemente.
6. Uma transação T não resultará uma operação `unlock(X)` a menos que ela já controle um bloqueio de leitura (compartilhado) ou um bloqueio de escrita (exclusivo) em um item X.

Conversão de Bloqueios. Às vezes é desejável relaxar as condições 4 e 5 da lista, de forma a permitir conversão de bloqueio, isto é, a uma transação que já controla um bloqueio no item X é permitido, sob certas condições, converter o bloqueio de um estado bloqueado para um outro. Por exemplo, é possível para uma transação T resultar em uma operação `read_lock(X)` e depois, com a promoção do bloqueio, gerar uma operação `write_lock(X)`. Se T é apenas a transação que controla um bloqueio de leitura em X no momento em que ela resulta na operação `write_lock(X)`, pode-se fazer a promoção do bloqueio; de outra forma, a transação deve esperar. Também é possível uma transação T produzir uma `write_lock(X)` e depois fazer um rebaixamento de bloqueio para uma operação `read_lock(X)`. Quando são usados a promoção e o rebaixamento

- 2 Esses algoritmos não permitem *promoção* ou *rebaixamento* de bloqueios, conforme descreveremos mais adiante nesta seção. O leitor pode estender os algoritmos para permitir essas operações adicionais.
- 3 Essa regra pode ser relaxada para permitir que uma transação desbloqueie um item e, então, o bloqueie novamente mais tarde.

422 Capítulo 18 Técnicas de Controle de Concorrência

de bloqueios, a tabela de bloqueio deve incluir identificadores de transação na estrutura de registro para cada bloqueio (no campo `transacao(oes)_bloqueio`, armazenar a informação sobre quais transações controlam os bloqueios no item). As descrições das operações `read_lock(X)` e `write_lock(X)`, na Figura 18.2, devem ser alteradas apropriadamente. Deixamos isso como um exercício para o leitor.

Read_lock(X):

```
B:ifLOCK(X)="unlocked" then begin LOCK(X)<- "read-locked"; no_of_reads(X)<-1 end else if LOCK(X)="read-locked"
then no_of_reads(X)<- no_of_reads(X) + 1 else begin wait (until LOCK(X)="unlocked" and
the lock manager wakes up the transaction); gotoB end;
```

write_lock(X):

```
B:ifLOCK(X)="unlocked" then LOCK(X)*- "write-locked"; else begin
wait (until LOCK(X)="unlocked" and
the lock manager wakes up the transaction); gotoB end;
```

unlock(X):

```
if LOCK(X)="write-locked" then begin LOCK(X)<~"unlocked";
wakeup one of the waiting transactions, if any end else if LOCK(X)="read-locked" then begin
no_of_reads(X)<- no_of_reads(X) - 1; if no_of_reads(X)=0 then begin LOCK(X)="unlocked";
wakeup one of the waiting transactions, if any end end;
```

FIGURA 18.2 Operações de bloqueio e desbloqueio para bloqueios de dois modos (leitura-escrita ou compartilhado-exclusivo).

Usar bloqueios binários ou de leitura/escrita nas transações, como descrito anteriormente, não *garante serialização* de planos de execução sobre si mesmos. A Figura 18.3 mostra um exemplo em que as regras de bloqueio anteriores são seguidas, mas podem produzir um plano de execução não serializável. Isso porque na Figura 18.3a os itens Y em T_1 e X em T_2 foram *desbloqueados muito cedo*, permitindo um plano de execução como o mostrado na Figura 18.3c, que não é serializado, e ainda fornecendo resultados incorretos. Para garantir a serialização, devemos seguir *um protocolo adicional*, relacionado ao posicionamento das operações de bloqueio e desbloqueio em toda transação. O melhor protocolo conhecido, em duas fases, será descrito na próxima seção.

18.1.2 Garantindo Serialização pelo Bloqueio em Duas Fases

Diz-se que uma transação segue o **protocolo de bloqueio em duas fases** se *todas* as operações (`read_lock`, `write_lock`) precedem a *primeira* operação de desbloqueio na transação. Tal transação pode ser dividida em duas fases: uma **fase de expansão** ou **crecimento (primeira)**, durante a qual novos bloqueios nos itens podem ser adquiridos, mas não podem ser liberados, e uma **fase de encolhimento (segunda)**, durante a qual os bloqueios existentes podem ser liberados, mas novos bloqueios não

4 Ela não está relacionada ao protocolo de efetivação em duas fases para recuperação em bancos de dados distribuídos (Capítulo 25).

1 8.1 Técnicas de Bloqueio em Duas Fases para Controle de Concorrência 423

podem ser adquiridos. Se a conversão de bloqueio for permitida, a promoção de bloqueios (da leitura-bloqueada para a escrita-bloqueada) deverá ser feita durante a fase de expansão, e o rebaixamento de bloqueios (da escrita-bloqueada para a leitura-bloqueada), na fase de encolhimento. Portanto, uma operação `read_lock(X)` que faz rebaixamento de um bloqueio de escrita já mantido em X pode aparecer apenas na fase de encolhimento.

As transações T_1 e T_2 da Figura 18.3a não seguem o protocolo de bloqueio em duas fases porque a operação `write_lock(X)` segue uma operação `unlock(Y)` em T_1 e, similarmente, a operação `write_lock(Y)` segue a operação `unlock(X)` em T_2 . Se impusermos um bloqueio em duas fases, as transações podem ser reescritas como T_1' e T_2' , como mostrado na Figura 18.4. Agora, o plano de execução mostrado na Figura 18.3c não é permitido para T_1' e T_2' (com sua ordem de operações de bloqueio e desbloqueio modificada) sob as regras de bloqueio descritas na Seção 18.1.1, porque T_1' produzirá sua `write_lock(X)` antes que ela desbloqueie o item Y; conseqüentemente, quando T_2' tenta sua `read_lock(X)`, ela é forçada a esperar até que T_1' libere o bloqueio, produzindo um `unlock(X)` no plano de execução.

(a)

```
T,
Read_lock(y);
Read_item(V);
unlock(V);
write_lock(X);
read_item(X);
X:=X+Y;
write_item(X);
unlock(X);
read_lock(X);
read_item(X);
unlock(X);
write_lock(V);
read_item(y);
Y:=X+Y;
write_item(Y);
unlock(V);
```

(b) Valores iniciais: X=20, V=30

Resultado do plano de execução serial T_1 seguido por T_2 :

X=50, V=80 Resultado do plano serial T_2 seguido por T_1 :

X=70, V=50

(c)

Tempo

```
read_lock(V); readItem(V); unlock(V);
read_lock(X);
read_item(X);
unlock(X);
write_lock(V);
read_item(y);
Y:=X+Y;
write_item(y);
unlock(y);
Resultado do plano S: X=50, y=50 (não serializavel)
write_lock(X);
read_item(X);
X:=X+;
write_item(X);
unlock(X);
```

FIGURA 18.3 Transações que não obedecem ao bloqueio em duas fases, (a) Duas transações T_1 e T_2 . (b) Resultado dos possíveis planos de execução seriais de T_1 e T_2 . (c) Um plano de execução não serializavel S que usa bloqueios.

```
read_kx*(V); readjem (Y); writejock (X); unkkx(y); read_item (X); X:=X+V; write_item (X); unlock (X);
read_lock (X); read_item (X); writejock (V); unlock (X); read_item (/); V:=X+V; write_item(y); unlock (Y);
```

FIGURA 18.4 Transações T_1' e T_2' , que são as mesmas que T_1 e T_2 da Figura 18.3, mas que seguem o protocolo de bloqueio de duas fases. Observe que elas podem produzir um deadlock.

424 Capítulo 18 Técnicas de Controle de Concorrência

Pode ser provado que, se *toda* transação em um plano de execução seguir o protocolo de bloqueio em duas fases, é *garantido* que o plano de execução *seja serializável*, evitando definitivamente a necessidade de testes para a serialização. O mecanismo de bloqueio, pela imposição das regras de bloqueio em duas fases, também impõe a serialização.

O bloqueio em duas fases pode limitar a quantidade de concorrência que pode surgir em um plano de execução. Isso porque uma transação T pode não estar apta a liberar um item X depois que ela o tiver usado completamente, se T deve bloquear um item adicional Y depois; ou, contrariamente, T deve bloquear o item adicional Y antes que ela necessite dele, de forma que possa liberar X. Portanto, X deve permanecer bloqueado por T até que todos os itens que a transação necessita ler ou escrever tenham sido bloqueados; só depois X pode ser liberado por T. Enquanto uma outra transação que procure acessar Y é forçada a esperar, ainda que T utilize X; da mesma forma, se Y for bloqueado antes do necessário, uma outra transação procurando acessar Y seria forçada a esperar, ainda que T não estivesse usando Y. Esse é o preço para garantir serialização de todos os planos, sem que eles mesmos tenham de fazer a verificação.

Bloqueio em Duas Fases Básico, Conservador, Estrito e Rigoroso. Há um número de variações de bloqueio em duas fases (2PL — *two-phase locking*). A técnica descrita há pouco é conhecida como 2PL básico. Uma variação conhecida como 2PL conservador (ou 2PL estático) requer uma transação para bloquear todos os itens que ela acessa antes *de a transação iniciar a execução*, pela pré-declaração de suas *read-set* e *write-set*. Recordamos da Seção 17.1.2 que a *read-set* de uma transação é o conjunto de todos os itens que a transação lê, e a *write-set* é o conjunto de todos os itens que ela grava. Se algum dos itens pré-declarados não precisar ser bloqueado, a transação não bloqueia nenhum item; ao contrário, ela espera até que todos estejam disponíveis para bloqueio. O 2PL conservador é um protocolo deadlock-free, conforme veremos na Seção 18.1.3, quando discutiremos o problema de deadlock. Entretanto, é difícil usá-lo na prática por causa da necessidade da pré-declaração de *read-set* e *write-set*, que não é possível na maioria das situações.

Na prática, a variação mais popular do 2PL é o 2PL estrito, que garante planos estritos (Seção 17.4). Nessa variação, uma transação T não libera nenhum de seus bloqueios exclusivos (escrita) até que ela efetive ou aborte. Portanto, nenhuma outra transação pode ler ou escrever um item que seja escrito por T, a menos que T tenha efetivado, gerando um plano de execução estrito para recuperação. O 2PL estrito não é *deadlock-free*. Uma variação mais restritiva do 2PL estrito é o 2PL rigoroso, que também garante planos de execução estritos. Nessa variação, uma transação T não libera nenhum de seus bloqueios (exclusivo ou compartilhado) até depois que efetive ou aborte, assim, é mais fácil de implementar que o 2PL estrito. Observe a diferença entre o 2PL conservador e o rigoroso; o primeiro deve bloquear todos os seus itens *antes que ele inicie*, de maneira que, uma vez que a transação começa, ela está em sua fase de encolhimento, ao passo que o outro não desbloqueia nenhum de seus itens até *depois que ele termine* (efetivando ou abortando), de modo que a transação está em sua fase de expansão até seu final.

Em muitos casos, o subsistema de controle de concorrência responsabiliza-se, ele mesmo, por gerar as solicitações de *read_lock* e *write_lock*. Por exemplo, suponha que o sistema imponha o protocolo 2PL estrito. Então, quando a transação T resulta em uma *ler_item(X)*, o sistema chama a operação *read_lock(X)* de interesse de T. Se o estado de *LOCK(X)* é *write_locked* por alguma outra transação T', o sistema coloca T na fila de espera para o item X; caso contrário, ele concede a solicitação *read_lock(X)* e permite que a operação *ler_item(X)* de T execute. Por outro lado, se a transação T realiza uma *escrever_item(X)*, o sistema chama a operação *write_lock(X)* de interesse de T. Se o estado de *LOCK(X)* é *write_locked* ou *read_locked* por alguma outra transação T', o sistema coloca T na fila de espera para o item X; se o estado de *LOCK(X)* é *read_locked* e T é ela mesma a única transação que controla o bloqueio de leitura em X, o sistema promove o bloqueio para *write_locked* e permite a operação *escrever_item(X)* por T; finalmente, se o estado de *LOCK(X)* é desbloqueado, o sistema concederá a solicitação *write_lock(X)* e permitirá que a operação *escrever_item(X)* execute. Depois de cada ação, o sistema deve atualizar sua tabela de bloqueio apropriadamente.

Embora o protocolo de bloqueio em duas fases garanta a serialização (isto é, todo plano de execução permitido é serializável), ele não permite *todos* os planos de execução serializáveis *possíveis* (isto é, alguns planos serializáveis serão proibidos pelo protocolo). Além disso, o uso de bloqueios pode causar dois problemas adicionais: deadlock e starvation. Veremos esses problemas e suas soluções na próxima seção.

18.1.3 Lidando com Deadlock (Impasse) e Starvation (Inanição)

O deadlock ocorre quando *cada* transação em um conjunto de *duas ou mais transações* espera por algum item que esteja bloqueado por alguma outra transação T' no conjunto. Portanto, cada transação no conjunto está em uma fila de espera, aguardando por uma das outras transações do conjunto para liberar o bloqueio em um item. Um exemplo simples é mostrado na Figura 18.5, onde as duas transações, T₁' e T₂', são travadas indefinidamente (deadlock) em um plano de execução parcial;

18.1 Técnicas de Bloqueio em Duas Fases para Controle de Concorrência 425

T_1' está em uma fila de espera por X, que está bloqueado por T_2' , enquanto T_2' está em uma fila de espera por Y, que está bloqueado por T_1' . Entretanto, nem T_1' nem T_2' nem qualquer outra transação pode acessar os itens X e Y.

V read_lock(Y); read_item(Y); write_lock(X);	V read_lock(X); read_item(X); write_lock(Y);
--	--

FIGURA 18.5 Ilustração do problema de deadlock: plano de execução parcial de T_1' , T_2' que está em um estado de deadlock.

Protocolos de Prevenção de Deadlock. Uma forma de prevenir deadlock é usar um protocolo de prevenção de deadlock. Um protocolo de prevenção de deadlock, que é usado em bloqueio em duas fases conservador, requer que cada transação *bloqueie todos os itens de que ela necessita no avanço* (que geralmente não é uma suposição prática) — se nenhum dos itens pode ser obtido, nenhum será bloqueado. Mais exatamente, a transação espera e, então, tenta novamente bloquear todos os itens de que ela necessita. Essa solução, obviamente, limita a concorrência. Um segundo protocolo, que também limita a concorrência, envolve *ordenar todos os itens* no banco de dados e garantir que a transação que necessita de diversos itens os bloqueará de acordo com essa ordem. Isso exige que o programador (ou o sistema) esteja ciente da ordem de escolha dos itens, o que também não é prático no contexto de banco de dados.

Outros esquemas de prevenção de deadlock têm sido propostos, permitindo decidir o que fazer com uma transação envolvida em uma possível situação de deadlock: ela deveria ser bloqueada e ter de esperar ser abortada ou assumir e abortar uma outra transação? Essas técnicas usam o conceito de timestamp (marca de tempo) de transação $TS(T)$, que tem um identificador único designado para cada transação. Os timestamps são baseados na ordem em que as transações são iniciadas, portanto, se a transação T_1 se iniciar antes da transação T_2 , então $TS(T_1) < TS(T_2)$. Observe que a transação *mais velha* tem o *menor* valor de timestamp. Dois esquemas que previnem deadlock são chamados esperar-morrer (wait-die) e ferir-esperar (wound-wait). Suponha que a transação T_1 tente bloquear um item X, mas não esteja apta porque X está bloqueado por alguma outra transação T com um bloqueio conflitante. As regras seguidas por esses esquemas são as seguintes:

- Esperar-morrer: Se $TS(T_1) < TS(T)$, então (T_1 é mais velha que T) T_1 pode esperar; caso contrário (T_1 é mais nova que T) aborta T_1 (T_1 morre) e reinicia mais tarde *com o mesmo timestamp*.
- Ferir-esperar: Se $TS(T_1) < TS(T)$, então (T_1 é mais velha que T) aborta T (T_1 fere T) e reinicia mais tarde *com o mesmo timestamp*; caso contrário (T_1 é mais nova que T) T_1 espera.

No esperar-morrer, a uma transação mais velha é permitido esperar por uma mais nova, enquanto uma transação mais nova solicitando um item controlado por uma transação mais velha é abortada e reiniciada. A abordagem ferir-esperar faz o oposto: a uma transação mais nova é permitido esperar por uma outra mais velha, enquanto uma transação mais velha, solicitando um item controlado por uma transação mais nova, *apropria-se* da transação mais nova, abortando-a. Ambos os esquemas finalizam abortando a *mais jovem* das duas transações, que *pode estar envolvida* em um deadlock. Pode ser mostrado que essas duas técnicas são deadlock-free, uma vez que, em esperar-morrer, as transações apenas esperam as transações mais novas, de forma que nenhum ciclo é criado. Similarmente, em ferir-esperar, as transações apenas esperam as transações mais velhas, de forma que nenhum ciclo é criado. Entretanto, ambas as técnicas podem levar algumas transações a serem abortadas e reiniciadas desnecessariamente, uma vez que aquelas transações podem *nunca, realmente, causar um deadlock*.

Um outro grupo de protocolos que previnem deadlock não exige timestamps. São os algoritmos no waiting (NW) e cautious waiting (CW). No algoritmo no waiting (sem espera), se uma transação não estiver apta a obter um bloqueio, ela será imediatamente abortada e, então, reiniciada depois de um certo tempo de atraso, sem verificar se um deadlock realmente ocorrerá ou não. Porque esse esquema pode causar transações que abortam e reiniciam desnecessariamente, o algoritmo cautious waiting (espera cuidadosa) foi proposto para tentar reduzir o número de abortos/reinícios desnecessários. Suponha que a transação T, tente bloquear um item X, mas não está apta a fazê-lo porque X está bloqueado por alguma outra transação T_1 com um bloqueio conflitante. As regras de cautious waiting são as seguintes:

- 5 Esses protocolos geralmente não são usados na prática ou por causa de suposições não realistas ou porque geralmente causam sobrecarga (overhead). A detecção de deadlock e timeouts (veja abaixo) é mais prática.

Tempo

426 Capítulo 18 Técnicas de Controle de Concorrência

• Cautious waiting: Se T não está bloqueado (não está esperando por algum outro item bloqueado), então T_1 é bloqueado e poderá esperar; caso contrário, aborta T_1 .

Pode ser demonstrado que o cautious waiting é deadlock-free, considerando o tempo $\mathbb{E}(T)$ para o qual cada transação bloqueada T permaneceu bloqueada. Se as duas transações, T_i e T , forem bloqueadas, e T_1 esteja esperando por T , então $b(T_i) \cdot b(T)$, uma vez que T_1 pode esperar por T por um período quando T_1 não está bloqueado. Portanto, os tempos de bloqueio formam uma ordenação total para todas as transações bloqueadas, de forma que nenhum ciclo que cause deadlock pode ocorrer.

Deteção de Deadlock e Timeouts. Uma segunda — e mais prática — abordagem para lidar com deadlocks é detecção de deadlock, na qual o sistema verifica se um estado de deadlock realmente existe. Essa solução é atrativa se soubermos que existem poucas interferências entre as transações — isto é, se diferentes transações raramente acessam os mesmos itens ao mesmo tempo. Isso pode acontecer se as transações forem curtas e se cada transação bloquear apenas poucos itens, ou se a carga da transação for leve. Por outro lado, se as transações forem longas e cada transação usar muitos itens, ou se a transação for muito pesada, pode ser vantajoso usar um esquema de prevenção de deadlock.

Uma forma simples para detectar um estado de deadlock é construir e manter um grafo wait-for (espera-por). Um nó criado no grafo wait-for para cada transação que é executada correntemente. Quando uma transação T_f está esperando para bloquear um item X , que está bloqueado por uma transação T_1 , uma ligação direcionada ($T_f \rightarrow T_1$) é criada no grafo. Quando T_1 libera o(s) bloqueio(s) dos itens pelos quais T_f estava esperando, a linha direcionada é eliminada do grafo wait-for. Temos um estado de deadlock se, e apenas se, o grafo wait-for tiver um ciclo. Um problema com essa abordagem é o caso de determinar *quando* o sistema verificaria um deadlock. Podem ser utilizados critérios como o número de transações concorrentemente executadas ou o período de tempo que diferentes transações têm esperado para conseguir bloquear os itens. A Figura 18.5 mostra o grafo wait-for para o plano de execução (parcial) mostrado na Figura 18.5a. Se o sistema estiver em um estado de dead lock, algumas das transações que causam o deadlock devem ser abortadas. Escolher quais transações abortar é conhecido como seleção de vítimas. O algoritmo para seleção de vítima evitaria selecionar transações que já estão em execução por um longo tempo e que fizeram muitas atualizações em vez disso, seriam selecionadas transações que ainda não fizeram muitas alterações.

Outro esquema simples para lidar com deadlock é o uso de timeouts. Esse método é prático por causa de sua baixa sobrecarga e simplicidade. Nesse método, se uma transação esperar por um período mais longo que um período de timeout definido pelo sistema, ele assumirá que a transação pode entrar num impasse (deadlock) e a abortará — independentemente se ur deadlock realmente existe ou não.

Starvation (Inanição). Um outro problema que pode ocorrer quando usamos bloqueio é a starvation, que ocorre quando uma transação não pode continuar por um período indefinido de tempo, enquanto outras transações no sistema continuam normalmente. Isso pode ocorrer se o esquema de espera para itens bloqueados for parcial, dando prioridade a algumas transações sobre outras. Uma solução para a inanição é ter um esquema de espera imparcial, tal como uma fila onde o primeiro que chega é o primeiro a ser servido; as transações estão habilitadas a bloquear um item na seqüência em que elas solicitaram o bloqueio originalmente. Um outro esquema permite às transações terem prioridade sobre as outras, ou seja, aumenta a prioridade de uma transação quanto maior for seu tempo de espera, até que ela chegue à prioridade mais alta e prossiga. A inanição pode ocorrer por causa da seleção de vítimas se o algoritmo seleciona a mesma transação como vítima repetidamente causando, então, o aborto e nunca permitindo o término da execução. O algoritmo pode usar prioridades mais altas para transações que tenham sido abortadas múltiplas vezes, a fim de evitar esse problema. Os esquemas esperar-morrer e ferir-morrer discutidos anteriormente, evitam a inanição.

18.2 CONTROLE DE CONCORRÊNCIA BASEADO EM ORDENAÇÃO POR TIMESTAMP

O uso de bloqueios, combinado com o protocolo 2PL, garante a serialização do plano de execução. Os planos serializáveis produzidos pelo 2PL têm seus planos seriais equivalentes baseados na ordem na qual as transações em execução bloqueiam os itens de que elas necessitam. Se uma transação precisa de um item que já esteja bloqueado, ela pode ser forçada a esperar até que o item seja liberado. Uma abordagem diferente, que garante serialização, envolve usar timestamps de transação para ordenar a execução da transação em um plano de execução serial equivalente. Na Seção 18.2.1 discutiremos timestamps, e na Seção 18.2.2, como a serialização é imposta pela ordenação de transações baseada em seus timestamps.

18.2.1 Timestamps

Relembramos que um timestamp é um identificador único criado pelo SGBD para identificar uma transação. Em geral, os valores de timestamp são designados na ordem em que as transações são submetidas ao sistema; assim, um timestamp pode ser pensado como o *momento do início da transação*. Podemos nos referir ao timestamp de uma transação T como TS(T). As técnicas de controle de concorrência baseadas em ordenação por timestamp não usam bloqueios, portanto, *deadlocks nunca podem ocorrer*.

Timestamps podem ser gerados de diversas maneiras. Uma possibilidade é usar um contador que é incrementado cada vez que seu valor é designado para uma transação. Os timestamps da transação são numerados como 1,2,3,... nesse esquema. Um contador de computador tem um valor máximo finito, assim, o sistema deve, periodicamente, reiniciar o contador em zero quando nenhuma transação estiver em execução durante um curto período de tempo. Um outro meio de implementar timestamps é usar o valor de data/hora corrente do relógio do sistema e garantir que dois valores de timestamp não serão gerados numa mesma marca de tempo do relógio.

18.2.2 O Algoritmo de Ordenação por Timestamp

A idéia para este esquema é ordenar as transações baseadas em seus timestamps. Um plano de execução no qual as transações participam é, então, serializável, e o plano de execução serial equivalente tem as transações na seqüência dos valores de seu timestamp, sendo chamado de ordenação por timestamp (TO). Observe como ela difere do 2PL, na qual um plano de execução é serializável por ser equivalente a *algum* plano de execução serial permitido pelos protocolos de bloqueio. Entretanto, na ordenação por timestamp, o plano de execução é equivalente à *ordem serial particular* correspondente à ordem dos timestamps da transação. O algoritmo deve garantir que, para cada item acessado pelas *operações conflitantes* no plano de execução, a ordem na qual o item é acessado não viola a ordem de serialização. Para fazer isso, o algoritmo associa, a cada item X do banco de dados, dois valores de timestamp (TS):

1. Read_TS(X): O timestamp de leitura do item X; esse é o maior timestamp entre todos os timestamps de transações que tenham lido o item X com sucesso — isto é, $\text{read_TS}(X) = \text{TS}(T)$, onde T é a transação *mais nova* que tenha lido X com sucesso.

2. Write_TS(X): O timestamp de escrita do item X; esse é o maior de todos os timestamps de transações que tenham escrito o item X com sucesso — isto é, $\text{write_TS}(X) = \text{TS}(T)$, em que T é a transação *mais nova* que tenha escrito X com sucesso.

Ordenação por Timestamp Básica. Se alguma transação T tenta executar uma operação ler_item(X) ou uma escrever_item(X), o algoritmo básico TO compara o timestamp de T com read_TS(X) e write_TS(X) para certificar-se de que a ordem de timestamp da execução da transação não é violada. Se essa ordem for violada, a transação T então aborta e é re-submetida ao sistema como uma nova transação e com um *novo timestamp*. Se T for abortada e revertida, qualquer transação T1 que possa ter usado um valor escrito por T deve ser revertida também. Similarmente, qualquer transação T2 que possa ter usado um valor escrito por T1 também deve ser revertida, e assim por diante. Este efeito é conhecido como cascading roll-back (retorno em cascata) e é um dos problemas associados à TO básica, uma vez que os planos de execução produzidos podem não ser recuperáveis. Um *protocolo adicional* deve ser imposto para garantir que os planos de execução sejam recuperáveis sem cascatas ou estritos. Descrevemos aqui, primeiro, o algoritmo de TO básica. O algoritmo de controle de concorrência deve verificar se operações conflitantes violam a ordenação por timestamp nos dois casos seguintes:

1. A transação T tenta uma operação escrever_item(X):

a. Se $\text{read_TS}(X) > \text{TS}(T)$ ou se $\text{write_TS}(X) > \text{TS}(T)$, então T aborta e reverte, e a operação é rejeitada. Isso acontece porque alguma transação mais nova, com um timestamp maior que TS(T) — e, portanto, *depois* de T na ordenação por timestamp — já leu ou escreveu o valor do item X antes que T tivesse uma chance para escrever X, violando, então, a ordenação por timestamp.

b. Se a condição na parte (a) não ocorrer, então T executa a operação escrever_item(X) e grava write_TS(X) com TS(T).

2. Transação T tenta uma operação ler_item(X):

a. Se $\text{write_TS}(X) > \text{TS}(T)$, então T aborta, reverte e rejeita a operação. Isso ocorre porque alguma transação mais nova, com timestamp maior que TS(T) — e, portanto, *depois* de T na ordenação por timestamp — já teria escrito o valor no item X antes que T tivesse a chance de ler X.

b. Se $\text{write_TS}(X) < \text{TS}(T)$, então executa a operação ler_item(X) de T, e atualiza read_TS(X) com o *maior* TS(T) e o read_TS(X) corrente.

428 Capítulo 18 Técnicas de Controle de Concorrência

Portanto, quando o algoritmo de TO básica detecta duas *operações conflitantes* que ocorrem na ordem incorreta, ele rejeita a última das duas operações, abortando a transação que a escolheu. Os planos produzidos pela TO básica são, portanto, livres de estarem em conflito serializável, como no protocolo 2PL. Entretanto, alguns planos são possíveis sob cada protocolo e não são permitidos por outros. Assim, nenhum protocolo permite *todos os planos possíveis* serializáveis. Como mencionado anteriormente, o deadlock não ocorre com a ordenação por timestamp. Contudo, o reinício cíclico (e por isso, a inanição) pode ocorrer se uma transação for continuamente abortada e reiniciada.

Ordenação Estrita por Timestamp. Uma variação da TO básica, chamada TO estrita, garante que os planos de execução sejam estritos (para recuperação fácil) e (conflito) serializáveis. Nessa variação, uma transação T que execute um $\text{ler_item}(X)$ ou $\text{escrever_item}(X)$, tal que $\text{TS}(T) > \text{write_TS}(X)$, terá suas operações de leitura ou escrita *atrasadas* até que a transação T' , que *escreveu* o valor de X (por isso, $\text{TS}(T') = \text{write_TS}(X)$), tenha efetivado ou abortado. Para implementar esse algoritmo é necessário simular o bloqueio de um item X que tenha sido escrito pela transação T até T' ser efetivada ou abortada. Esse algoritmo não causa deadlock, uma vez que T espera por T' apenas se $\text{TS}(T) > \text{TS}(T')$.

Regra de Escrita de Thomas. Uma modificação do algoritmo de TO básica, conhecida como regra de escrita de Thomas, não impõe serialização de conflito, mas rejeita menos operações de escrita (*write*), modificando as verificações para a operação $\text{escrever_item}(X)$ como segue:

1. Se $\text{read_TS}(X) > \text{TS}(T)$, então aborta e reverte T e rejeita a operação.
2. Se $\text{write_TS}(X) > \text{TS}(T)$, então não executa a operação *write*, mas continua processando. Isso ocorre porque algumas transações com timestamp maior que $\text{TS}(T)$ — e, portanto, depois de T na ordenação por timestamp — já gravaram o valor de X . Portanto, devemos ignorar a operação $\text{escrever_item}(X)$ de T porque ela já está antiquada e obsoleta. Observe que qualquer conflito gerado por essa situação seria detectado pelo caso (1).
3. Se nem a condição na parte (1) nem a condição na parte (2) ocorrerem, então executa a operação de $\text{escrever_item}(X)$ de T e aponta $\text{write_TS}(X)$ para $\text{TS}(T)$.

18.3 TÉCNICAS DE CONTROLE DE CONCORRÊNCIA DE MULTIVERSÃO

Outros protocolos para controle de concorrência mantêm os valores antigos de um item de dado quando o item é atualizado. Eles são conhecidos como controle de concorrência de multiversão porque diversas versões (valores) de um item são mantidas. Quando uma transação exige acesso a um item, uma versão *apropriada* é escolhida para manter a serialização do plano em execução, se possível. A idéia é que algumas operações de leitura que foram rejeitadas em outras técnicas possam ser aceitas pela leitura de uma versão *mais antiga* do item, garantindo a serialização. Quando uma transação escreve um item, escreve uma *nova versão*, e a versão anterior do item é mantida. Alguns algoritmos de controle de concorrência de multiversão usam o conceito de visão serializada em vez de serialização de conflito.

Uma desvantagem óbvia das técnicas de multiversão é a necessidade de mais área para armazenar as múltiplas versões dos itens de banco de dados. Entretanto, versões anteriores podem ter de ser mantidas de qualquer maneira — por exemplo, para possibilitar a recuperação. Além disso, algumas aplicações de banco de dados exigem acesso a versões anteriores para acompanhamento do histórico da evolução dos valores dos itens de dados. O caso extremo é um *banco de dados temporal* (Capítulo 24), que mantém o controle de todas as mudanças e os tempos nos quais elas ocorreram. Nesses casos, não há penalidade de armazenamento adicional para as técnicas de multiversão, uma vez que versões mais antigas já são armazenadas.

Diversos esquemas de controle de concorrência de multiversão têm sido propostos. Veremos dois desses esquemas: um baseado em ordenação por timestamp e outro baseado no 2PL.

18.3.1 Técnica de Multiversão Baseada em Ordenação por Timestamp

Neste método, diversas versões X_1, X_2, \dots, X_n de cada item de dado X são armazenadas. Para *cada versão*, o valor da versão X , e os dois timestamps seguintes são mantidos:

1. $\text{read_TS}(X_i)$: O timestamp *read* de X_i é o maior de todos os timestamps das transações que tenham lido com sucesso a versão X_i .
2. $\text{writeTS}(X_i)$: O timestamp *write* de X_i é o timestamp da transação que escreveu o valor da versão X_i .

18.3 Técnicas de Controle de Concorrência de Multiversão 429

Se uma transação T pode executar uma operação `escrever_item(X)`, uma nova versão X_{k+1} , do item X será criada, com ambas, a `write_TS(Xk+1)` e a `read_TS(Xk+1)`, atualizadas para TS(T). Da mesma forma, quando uma transação T pode ler o valor da versão X_i , o valor de `read_TS(Xi)` será atualizado para o maior `read_TS(Xi)` e TS(T) correntes.

Para garantir a serialização, duas regras são usadas:

1. Se a transação T tenta executar uma operação `escrever_item(X)`, e a versão i de X tem a mais alta `write_TS(Xi)` de todas as versões de X, que também é *menor ou igual a* TS(T), e `read_TS(Xi) > TS(T)`, então ela aborta e reverte a transação T; caso contrário, criará uma nova versão X de X com `read_TS(Xi) = write_TS(X) = TS(T)`.

2. Se a transação T tenta executar uma operação `ler_item(X)`, encontra a versão i de X que tem a mais alta `write_TS(Xi)` de todas as versões de X, que também é *menor ou igual a* TS(T); então retorna o valor de X_i para a transação T e atualiza o valor de `read_TS(Xi)` para o maior TS(T) e `read_TS(Xi)` corrente.

Como podemos ver no caso 2, uma `ler_item(X)` é sempre bem-sucedida, uma vez que encontra a versão apropriada de X, para ler, baseada na `write_TS` das várias versões existentes de X. No caso 1, entretanto, a transação T pode ser abortada e revertida. Isso acontece se T tentar gravar uma versão de X que teria sido lida por uma outra transação T', cujo timestamp seja `read_TS(Xi)`; entretanto, T' já leu a versão X_i , que foi gravada pela transação com timestamp igual a `write_TS(Xi)`. Se ocorrer esse conflito, T é revertida; caso contrário, uma nova versão de X, gravada pela transação T, é criada. Observe que, se T for revertida, pode ocorrer um retorno em cascata (cascading rollback). Portanto, para garantir a recuperação, uma transação T não deveria ser efetivada até que todas as transações que tenham gravado alguma versão lida por T tivessem sido efetivadas.

18.3.2 Bloqueio em Duas Fases de Multiversão Usando Bloqueios de Certificação

Neste esquema de bloqueio múltiplo-modo, há três *modos de bloqueio* para um item: ler, gravar e certificar, em vez de apenas os dois modos (ler, gravar) vistos anteriormente. Portanto, o estado de LOCK(X) para um item X pode ser um `read_locked` (leitura bloqueada), `write_locked` (escrita bloqueada), `certify_locked` (certificação bloqueada) ou `unlocked` (desbloqueado). No esquema de bloqueio padrão apenas com bloqueios de leitura e escrita (Seção 18.1.1), um bloqueio de escrita (write) é exclusivo. Podemos descrever o relacionamento entre bloqueios de leitura e escrita no esquema padrão pelo significado da tabela de compatibilidade de bloqueio, mostrada na Figura 18.6a. Uma entrada de *sim* significa que, se uma transação T controla o tipo de bloqueio especificado no cabeçalho da coluna do item X, e se a transação T solicita o tipo de bloqueio especificado no cabeçalho da linha do mesmo item X, então T' pode *obter o bloqueio* porque os modos de bloqueio são compatíveis. Por outro lado, uma entrada *não* na tabela indica que os bloqueios não são compatíveis, assim, T' deve esperar até que T libere o bloqueio.

No esquema de bloqueio padrão, uma vez que uma transação obtém um bloqueio de escrita em um item, nenhuma outra transação pode acessar aquele item. A idéia por trás do 2PL multiversão é permitir que outras transações T' leiam um item X enquanto uma transação T única controla um bloqueio de escrita em X. Isso é possível utilizando-se *duas versões* para cada item X; uma versão deve sempre ser gravada por alguma transação efetivada. A segunda versão X' é criada quando uma transação T adquire um bloqueio em um item. Outras transações podem continuar para ler a *versão efetivada* de X, enquanto T controla o bloqueio de escrita. A transação T pode gravar o valor de X' conforme a necessidade, sem afetar o valor da versão X efetivada. Entretanto, uma vez que T esteja pronta para efetivar, ela deve obter um bloqueio de certificação de todos os itens que ela controla correntemente de bloqueios de escrita, antes que ela efetive. O bloqueio de certificação não é compatível com os bloqueios de leitura, assim, a transação pode ter de atrasar sua efetivação até que todos os seus itens de escrita bloqueados sejam liberados por quaisquer transações, no sentido de obter os certificados de bloqueio. Uma vez que os certificados — cujos bloqueios são exclusivos — são adquiridos, a versão X efetivada do item de dado é apontada para o valor da versão X' , que é descartada, e os bloqueios de certificação são, então, liberados. A tabela de compatibilidade de bloqueio para esse esquema é mostrado na Figura 18.6b.

Nesse esquema 2PL de multiversão, as leituras podem continuar concorrentemente com uma única operação de escrita — um arranjo não permitido sobre os esquemas padrão 2PL. O custo é que uma transação pode ter de atrasar até obter bloqueios de certificação exclusivos *de todos os itens* atualizados. Pode ser demonstrado que esse esquema evita abortos em cascata, uma vez que as transações podem ler a versão X que foi gravada por uma transação efetivada. Entretanto, podem ocorrer deadlocks se a promoção de um bloqueio de leitura para um bloqueio de escrita for permitido, e este deve ser controlado pelas variações das técnicas discutidas na Seção 18.1.3.

430 Capítulo 18 Técnicas de Controle de Concorrência

(a)

Ler Escrever

Ler

Escrever

nao

nao

(b)

Ler Escrever Certificação

Ler

Escrever

Certificação

sim

nao

nao

nao

FIGURA 18.6 Tabelas de compatibilidade de bloqueio, (a) Uma tabela de compatibilidade para o esquema de bloque read/write. (b) Uma tabela de compatibilidade para o esquema de bloqueio ler/escrever/certificar.

18.4 TÉCNICAS DE CONTROLE DE CONCORRÊNCIA DE VALIDAÇÃO (OTIMISTA)

Em todas as técnicas de controle de concorrência discutidas até aqui, sempre se faz alguma verificação *antes* que uma operação de banco de dados possa ser executada. Por exemplo, é feita uma verificação no bloqueio para determinar se o item que está sendo acessado está bloqueado. Na ordenação por timestamp, o timestamp da transação é verificado em relação aos timestamps de leitura e escrita do item. Tal verificação representa sobrecarga durante a execução da transação, com o efeito de reduzir a velocidade das transações.

Nas técnicas de **controle de concorrência otimista**, também conhecidas como **validação** ou **técnicas de certificação**, *nenhuma verificação* é feita enquanto a transação estiver sendo executada. Diversos métodos de controle de concorrência propostos usam a técnica de validação. Descreveremos apenas um esquema em que as atualizações da transação *não* são aplicadas diretamente aos itens de banco de dados até que a transação alcance seu fim. Durante sua execução, todas as atualizações são aplicadas às *cópias locais* dos itens de dados, os quais são mantidos para as transações. Ao fim da execução, uma **fase de validação** verifica se alguma das atualizações de transações violou a serialização. Certas informações utilizadas pela fase de validação devem ser mantidas pelo sistema. Se a serialização não for violada, a transação é efetivada, e o banco de dados é atualizado com as cópias locais; caso contrário, a transação é abortada e, então, reiniciada mais tarde.

Há três fases para esse protocolo de controle de concorrência:

1. **Fase de leitura:** Uma transação pode ler valores de itens de dados efetivados do banco de dados. Entretanto, as atualizações são aplicadas apenas às cópias locais (versões) dos itens de dados mantidos no espaço de trabalho da transação

2. **Fase de validação:** A verificação é executada para garantir que a serialização não seja violada se as atualizações da transação forem aplicadas ao banco de dados.

3. **Fase de escrita:** Se a fase de validação é bem-sucedida, as atualizações da transação são aplicadas ao banco de dados; caso contrário, as atualizações são descartadas e a transação é reiniciada.

A idéia contida no controle de concorrência otimista é fazer todas as verificações de uma vez, portanto, a execução de transação continua com um mínimo de sobrecarga até que a fase de validação seja alcançada. Se há pouca interferência entre as transações, a maioria será validada com sucesso. Entretanto, se há muita interferência, várias transações em execução terão seus resultados descartados, e deverão ser reiniciadas mais tarde. Sob essas circunstâncias, as técnicas otimistas não trabalham bem. Elas são chamadas de 'otimistas' porque assumem que ocorrerá pouca interferência e, em consequência, não é necessário fazer verificação durante a execução da transação.

O protocolo otimista que descrevemos usa timestamps de transação e também exige que os `wri te_sets` (conjuntos_escrita) e `read_sets` (conjuntos_leitura) das transações sejam mantidos pelo sistema. Além disso, os tempos de *início* e *fim* para algumas das três fases necessitam ser mantidos para cada transação. Recordamos que o `wri te_set` (**conjunto_escrita**) de um

6 Note que isso pode ser considerado como armazenamento de múltiplas versões de itens!

18.5 Granularidade de Itens de Dados e Bloqueio de Granularidade Múltipla 431

transação é o conjunto dos itens que ela escreve, e o **read_set** (conjunto_leitura) é o conjunto de itens que ela lê. Na fase de validação para a transação T_f o protocolo verifica se T_f não interfere em quaisquer transações efetivadas ou em quaisquer outras transações correntes em sua fase de validação. A fase de validação para T , verifica que, para *cada* transação "L que é efetivada ou está em sua fase de validação, ocorre *uma* das seguintes condições:

1. A transação T completa sua fase de escrita antes que T_f inicie sua fase de leitura.
2. T_f inicia sua fase de escrita depois que T completa sua fase de escrita, e o **read_set** de T_f não tem itens em comum com o **wri te_set** de T_f .
3. Ambos, o **read_set** e o **wri te_set** de T_f não têm itens em comum com o **wri te_set** de T , e T completa sua fase de leitura antes que T_f complete a sua.

Quando a transação T_f está em fase de validação, a condição (1) é verificada primeiro para cada transação T , uma vez que é a condição mais simples. Apenas se a condição (1) é falsa, a condição (2) é verificada, e apenas se (2) é falsa, a condição (3) — a mais complexa para avaliar — é verificada. Se nenhuma dessas três condições acontece, não há interferência, e T_f é validada com sucesso. Se nenhuma dessas três condições acontece, a validação da transação T_f falha e ela é abortada e reiniciada mais tarde, por causa da interferência que *pode* ocorrer.

(corrigir – redigitar e verificar os índices de T acima)

18.5 GRANULARIDADE DE ITENS DE DADOS E BLOQUEIO DE GRANULARIDADE MÚLTIPLA

Todas as técnicas de controle de concorrência assumem que o banco de dados é formado por um conjunto de itens de dados nomeados. Um item do banco de dados pode ser:

- Um registro de banco de dados.
- Um valor de campo de um registro de banco de dados.
- Um bloco de disco.
- Um arquivo.
- Um banco de dados inteiro.

A granularidade pode afetar a execução do controle de concorrência e recuperação. Na Seção 18.5.1, discutiremos algumas das decisões relativas à escolha do nível de granularidade usado para o bloqueio e, na Seção 18.5.2, veremos um esquema de bloqueio de granularidade múltipla, no qual o nível de granularidade (tamanho do item de dado) pode ser alterado dinamicamente.

18.5.1 Considerações sobre Níveis de Granularidade para Bloqueio

O tamanho dos itens de dados é, costumeiramente, chamado de **granularidade do item de dados**. A *granularidade fina* refere-se aos tamanhos pequenos de item, enquanto a *granularidade grossa* refere-se aos tamanhos grandes de item. Diversas decisões devem ser tomadas na escolha do tamanho do item de dado. Discutiremos tamanho de item de dado no contexto de bloqueio, embora argumentos similares possam ser utilizados para outras técnicas de controle de concorrência.

Primeiro, observe que, quanto maior o tamanho do item de dados, mais baixo é o grau de concorrência permitido. Por exemplo, se o tamanho do item de dados é um bloco de disco, uma transação T que precisa bloquear um registro B deve bloquear todo o bloco X de disco que contém B porque um bloqueio está associado a todo o item de dados (bloco). Agora, se uma outra transação S quer bloquear um registro C , que esteja armazenado no mesmo bloco X em um modo de bloqueio conflitante, ela é forçada a esperar. Se o tamanho do item de dados fosse um registro único, a transação S estaria apta a continuar porque estaria bloqueando um item de dados diferente (registro).

Porém, quanto menor o tamanho do item de dados, maior será o número de itens em um banco de dados. Como cada item está associado a um bloqueio, o sistema terá um número maior de bloqueios ativos a serem controlados pelo gestor de bloqueios. Mais operações de bloqueio e desbloqueio serão executadas, causando uma sobrecarga mais alta. Além disso, mais espaço de armazenamento será exigido para a tabela de bloqueios. Para timestamps, o armazenamento é exigido para a **read_TS** e para a **write_TS** de cada item de dado, e existirá sobrecarga similar para manipular um número de itens grande.

Considerando as opções acima, uma questão óbvia pode ser colocada: Qual é o melhor tamanho de item? A resposta é: *depende dos tipos de transações envolvidas*. Se uma transação típica acessa um número pequeno de registros, é vantajoso ter a granularidade do item de dados em um registro. Por outro lado, se uma transação típica acessa muitos registros em um mesmo arquivo, pode ser melhor usar a granularidade de bloco ou de arquivo, de forma que a transação irá considerar todos aqueles registros como um (ou poucos) item de dados.

18.6 Usando Bloqueios para Controle de Concorrência em Índices 433

5. Uma transação T pode bloquear um nó apenas se ela não tiver nenhum nó desbloqueado (para garantir o protocolo 2PL).
 6. Uma transação T pode desbloquear um nó N apenas se nenhum dos nós filhos de N estiverem correntemente bloqueados por T. A regra 1 garante que bloqueios conflitantes não podem ser concedidos. As regras 2, 3 e 4 declaram as condições quando uma transação pode bloquear um dado nó em qualquer um dos modos de bloqueio. As regras 5 e 6 do protocolo MGL garantem que as regras do 2PL produzem planos serializáveis. Para ilustrar o protocolo MGL com a hierarquia de banco de dados na Figura 18.7, considere as três transações seguintes:

1. T₁ quer atualizar os registros der_{in} até $r_{2i}j$.
2. T₂ quer atualizar todos os registros na página p_{12} .
3. T₃ quer ler o registro r, j e o arquivo f_2 .

A Figura 18.9 mostra um plano de execução serializável possível para essas três transações. Apenas as operações de bloqueio são mostradas. A notação <tipo_bloqueio (<item>) é usada para exibir as operações de bloqueio no plano de execução.

IS IX

S

SIX

X FIGURA 18.8 Uma matriz de compatibilidade de bloqueio para bloqueio de granularidade múltipla.

O protocolo de nível de granularidade múltipla é especialmente adequado quando processa uma mistura de transações que incluem: (1) transações curtas que acessam poucos itens (registros ou campos) e (2) transações longas que acessam arquivos. Nesse ambiente ocorrem menos bloqueios de transações e menos overhead de bloqueio com esse protocolo quando comparado a uma abordagem de nível único de bloqueio de granularidade.

18.6 USANDO BLOQUEIOS PARA CONTROLE DE CONCORRÊNCIA EM ÍNDICES

O bloqueio em duas fases também pode ser aplicado para índices (Capítulo 14), onde os nós de um índice correspondem às páginas de disco. Entretanto, bloqueios controlados em páginas de índice até a fase de encolhimento do 2PL poderiam causar um montante indevido de bloqueios de transações, porque a pesquisa de um índice sempre *inicia na raiz*; assim, se uma transação quiser inserir um registro (operação de escrita), a raiz deveria ser bloqueada em modo exclusivo; desse modo, todos os outros bloqueios conflitantes solicitados para o índice devem esperar até que a transação entre em sua fase de encolhimento. Assim ocorre a interrupção de todas as outras transações que acessariam o índice, de forma que, na prática, devem ser utilizadas outras abordagens para o bloqueio de índice.

A estrutura de árvore de índice pode ser vantajosa quando do desenvolvimento de um esquema de controle de concorrência. Por exemplo, quando uma pesquisa no índice (operação de leitura) está sendo executada, um caminho na árvore é percorrido, da raiz à folha. Uma vez que um nó de nível mais baixo no caminho tenha sido acessado, os nós de nível mais alto nesse caminho não seriam usados novamente. Assim, uma vez que um bloqueio de leitura em um nó filho é obtido, o bloqueio no pai pode ser liberado. Também quando uma inserção está sendo aplicada a um nó folha (isto é, quando uma chave e um ponteiro são inseridos), então o nó específico precisa ser bloqueado em modo exclusivo. Entretanto, se esse nó não está cheio, a inserção não causará mudanças aos nós de índices de nível mais alto, o que implica que eles não necessitam estar bloqueados exclusivamente.

IS	IX	S	SIX	X
sim	sim	sim	sim	não
sim	sim	não	não	não
sim	não	sim	não	não
sim	não	não	não	não
não	não	não	não	não

434 Capítulo 18 Técnicas de Controle de Concorrência

T ₁	T ₂	T ₃
IX(db)		
IX(cto)		
IS(db) IS(f,) IS(P,,) IX(P,,)		
IX(/,)		
X(p ₁₂)		
S(r ₁₁ ,) IX(Q IX(p ₂₁) X(r ₂₁₁))		
unlock(r ₂₁₁) unlock(p ₂ ,) unlock(f ₂)		
S(f ₂) unlock(p ₁₂) unlock^ unlock(db) unlockí,,,) unlock(p,,) unlock(f,) unlock(db)		
unlock(r ₁₁ ,)		
unlockí,,)		
unlock(f,)		
unlock(f ₂)		
unlock(dd)		

FIGURA 18.9 Operações de bloqueio para ilustrar um plano de execução serializável.

Uma abordagem conservadora para inserções seria bloquear o nó raiz em modo exclusivo e, então, acessar o nó filho apropriado da raiz. Se o nó filho não estiver cheio, então o bloqueio no nó raiz poderá ser liberado. Essa abordagem pode ser aplicada em todo o caminho, da árvore à folha, que são, normalmente, três ou quatro níveis da raiz. Embora bloqueios exclusivos sejam rígidos, eles são prontamente liberados. Uma abordagem alternativa mais otimista seria solicitar e compartilhar bloqueios controlados nos nós que levam ao nó folha, com bloqueio exclusivo na folha. Se a inserção causar divisão na folha, a inserção se propagaria para nó(s) de nível mais alto. Então, os bloqueios nos nós de nível mais alto deveriam ser no modo exclusivo.

Uma outra abordagem para bloqueio de índice é usar uma variante da árvore B, chamada árvore de ligação-E (B-link). Em uma árvore de ligação-B, nós irmãos no mesmo nível são conectados em cada nível, permitindo que bloqueios compartilhados sejam usados quando solicitarem uma página e exigindo que o bloqueio seja liberado antes de acessar o nó filho. Para uma operação de inserção, o bloqueio compartilhado em um nó seria promovido a um modo exclusivo. Se ocorrer divisão, o nó pai deve ser bloqueado novamente em modo exclusivo. Há uma complicação quando ocorrem operações de pesquisa executadas concorrentemente com a atualização. Suponha que uma operação de atualização concorrente siga o mesmo caminho que a busca e insira uma nova entrada na folha nó. Além disso, suponha que a inserção cause divisão no nó da folha. Quando a inserção é feita, o processo de busca recomeça, direcionando o ponteiro para a folha desejada apenas para descobrir que aquela chave que ele está procurando não está presente porque a divisão moveu a chave para um novo nó folha, provavelmente o irmão *da direita* do nó folha original. Entretanto, o processo de busca ainda pode continuar se ele seguir o ponteiro (ligação) do nó folha original para seu irmão à direita, onde está a chave desejada.

A manipulação do caso de remoção, no qual dois ou mais nós da árvore de índice se unem, também é parte do protocolo de concorrência da árvore de ligação-B. Nesse caso, bloqueios nos nós a serem unidos são efetuados, assim como o bloqueio no pai dos dois nós a serem unidos.

18.7 OUTROS TÓPICOS SOBRE CONTROLE DE CONCORRÊNCIA

Nesta seção, discutiremos algumas opções relevantes para o controle de concorrência. Na Seção 18.7.1 analisaremos os problemas associados à inserção e remoção de registros, e o problema conhecido como *problema do fantasma*, que pode ocorrer quando registros são inseridos. Ele foi descrito como um problema potencial que exigiria uma medida de controle de concorrência na Seção 17.6. Na Seção 18.7.2 veremos problemas que podem ocorrer quando uma transação produz dados para um monitor antes de efetivá-los, então a transação é abortada mais tarde.

18.7.1 Inserção, Remoção e Registros Fantasmas

Quando um novo item de dado é **inserido** no banco de dados, é óbvio que ele não pode ser acessado até depois que o item seja criado e a operação de inserção seja completada. Em um ambiente de bloqueio, um bloqueio para o item pode ser criado e definido no modo exclusivo (escrita); o bloqueio pode ser liberado ao mesmo tempo que outros bloqueios de escrita são liberados, com base no protocolo de controle de concorrência que esteja sendo usado. Para um protocolo baseado em timestamp, os timestamps de leitura e escrita do novo item têm o timestamp da transação de criação.

A seguir, considere a operação de remoção que é aplicada a um item existente. Para protocolos de bloqueio, novamente um protocolo exclusivo (de escrita) deve ser obtido antes que a transação possa remover o item. Para ordenação por timestamp, o protocolo deve garantir que nenhuma transação atrasada tenha lido ou escrito o item antes de ele ser removido.

Uma situação conhecida como o **problema do fantasma** pode ocorrer quando um novo registro que está sendo inserido por alguma transação T satisfaz a condição que um conjunto de registros acessados por uma outra transação T' deveria satisfazer. Por exemplo, suponha que a transação T esteja inserindo um novo registro de EMPREGADO, cujo DNO = 5, enquanto a transação T' está acessando todos os registros de EMPREGADO, cujos DNO = 5 (quer dizer, adicionar todos os seus valores de SALÁRIO para calcular o orçamento da equipe para o departamento 5). Se a ordem serial equivalente for T seguida por T₁ então T deve ler o novo registro de EMPREGADO e incluir seu SALÁRIO no cálculo da soma. Para a ordem serial equivalente T seguida por T₁ o novo salário não seria incluído. Observe que, embora as transações conflitem logicamente, no último caso realmente não há registro (item de dado) comum entre as duas transações, uma vez que T pode ter bloqueado todos os registros com DNO = 5 *antes* que T insira o novo registro. Isso ocorre porque o registro que causa o conflito é um **registro fantasma**, que aparece subitamente no banco de dados. Se outras operações nas duas transações conflitarem, o conflito causado pelo registro fantasma pode não ser reconhecido pelo protocolo de controle de concorrência.

Uma solução para o problema do registro fantasma é usar o **bloqueio de índice**, como discutido na Seção 18.6. Recordamos do Capítulo 14 que um índice inclui entradas que têm um valor de atributo, mais um conjunto de ponteiros para todos os registros no arquivo com esse valor. Por exemplo, um índice em DNO de EMPREGADO incluiria uma entrada para cada valor distinto de DNO, mais um conjunto de ponteiros para todos os registros de EMPREGADO com este valor. Se a entrada de índice for bloqueada antes que o registro possa ser acessado, então o conflito do registro fantasma pode ser detectado. Isso ocorre porque a transação T' solicitaria um bloqueio de leitura na *entrada de índice* para DNO = 5, e T solicitaria um bloqueio de escrita na mesma entrada antes que se coloque o bloqueio nos registros. Uma vez que o bloqueio de índice gera conflito, o conflito do fantasma seria detectado. Uma técnica mais geral, chamada **bloqueio de predicado**, bloquearia o acesso a todos os registros que satisfazem um *predicado arbitrário* (condição) de uma maneira similar; entretanto, bloqueios de predicado têm se mostrado difíceis de implementar eficientemente.

18.7.2 Transações Interativas

Outro problema ocorre quando transações interativas lêem uma entrada e escrevem uma saída em um dispositivo interativo, tal como um monitor de tela, antes que elas sejam efetivadas. O problema é que um usuário pode entrar com um valor de um item de dado para uma transação T com base em um valor mostrado na tela pela transação T', que pode não ter sido efetivada. Essa dependência entre T e T' não pode ser modelada pelo método de controle de concorrência do sistema, uma vez que está baseada na interação do usuário com as duas transações.

Uma abordagem para tratar esse problema é adiar a saída de transações para a tela até que ela tenha se efetivado.

18.7.3 Travas

Bloqueios de curta duração são chamados de **travas**. As travas não seguem os protocolos usuais de controle de concorrência, tal como o bloqueio de duas fases. Por exemplo, uma trava pode ser usada para garantir a integridade física de uma página

quando esta está sendo escrita do *buffer* para o disco. Uma trava seria fornecida para a página, a página escrita para o disco e, então, a trava seria liberada.

18.8 RESUMO

Neste capítulo vimos as técnicas de SGBD para controle de concorrência. Começamos discutindo os protocolos baseados em bloqueio, que são os mais comumente usados na prática. Descrevemos o protocolo de bloqueio em duas fases (2PL) e algumas de suas variações: o 2PL básico, o 2PL estrito, o 2PL conservador e o 2PL rigoroso. As variações estrita e rigorosa são as mais comuns por causa de suas propriedades de recuperação, que são melhores. Introduzimos os conceitos de bloqueio compartilhado (leitura) e exclusivo (escrita) e mostramos como o bloqueio pode garantir a serialização quando usado em conjunção com a regra de bloqueio em duas fases. Apresentamos também várias técnicas para tratar o problema de deadlock, que pode ocorrer com o bloqueio. Na prática, é comum usar timeouts (limites de tempo) e detecção de deadlock (grafos espera-por).

Depois apresentamos outros protocolos de controle de concorrência que não são usados frequentemente na prática, mas são importantes dadas as alternativas teóricas que propõem para solucionar esse problema. Elas incluem o protocolo de ordenação por timestamp (marca de tempo), que garante a serialização baseada na ordem dos timestamps das transações. Timestamps são identificadores únicos de transação gerados pelo sistema. Discutimos a regra de escrita de Thomas, que melhora a execução, mas não garante a serialização de conflito. O protocolo de ordenação por timestamp estrito também foi apresentado. Depois analisamos dois protocolos de multiversão que assumem que as versões mais antigas dos itens de dados podem ser mantidas no banco de dados. Uma técnica, chamada bloqueio de multiversão em duas fases (que também tem sido usada na prática), assume que podem existir duas versões para um item e tenta aumentar a concorrência, tornando os bloqueios de escrita e leitura compatíveis (pelo custo da introdução de um modo de bloqueio de certificação adicional). Apresentamos, também, um protocolo de multiversão baseado na ordenação por timestamp. Em seguida, um exemplo de um protocolo otimista, também conhecido como protocolo de certificação ou validação.

Então voltamos nossa atenção para a importante definição prática da granularidade do item de dado. Descrevemos um protocolo de bloqueio de multigranularidade, que permite a mudança da granularidade (tamanho do item), com base nas diferentes transações em execução, com o objetivo de melhorar a performance do controle de concorrência. Outra alternativa prática importante foi apresentada, que é o desenvolvimento de protocolos de bloqueio para índices, de maneira que os índices não se tornem impedimento ao acesso concorrente. Finalmente, introduzimos o problema do fantasma e problemas com transações interativas, e descrevemos rapidamente o conceito de travas e como elas diferem dos bloqueios.

No próximo capítulo, daremos uma visão das técnicas de recuperação.

Questões de Revisão

- 18.1. O que é o protocolo de bloqueio em duas fases? Como ele garante a serialização?
- 18.2. Quais são algumas das variações do protocolo de bloqueio em duas fases? Porque, frequentemente, é preferível o protocolo em duas fases estrito ou rigoroso?
- 18.3. Discuta os problemas de deadlock (impasse) e starvation (inanição) e as diferentes abordagens para a manipulação desses problemas.
- 18.4. Compare os bloqueios binários com os bloqueios exclusivos/compartilhados. Por que o último tipo de bloqueio é preferível?
- 18.5. Descreva os protocolos esperar-morrer e ferir-esperar para prevenção de deadlock.
- 18.6. Descreva os protocolos de espera cuidadosa (cautious waiting), sem espera (no waiting) e timeout para prevenção de deadlock.
- 18.7. O que é um timestamp? Como o sistema gera timestamps?
- 18.8. Discuta o protocolo de ordenação por timestamp para controle de concorrência. Como a ordenação estrita por timestamp difere da ordenação básica por timestamp?
- 18.9. Discuta duas técnicas de multiversão para controle de concorrência.
- 18.10. O que é um protocolo de certificação? Quais são as vantagens e as desvantagens de usar bloqueios de certificação?
- 18.11. Como as técnicas de controle de concorrência otimistas diferem das outras técnicas de controle de concorrência? Por que elas são chamadas de técnicas de validação ou certificação? Discuta as fases típicas de um método de controle de concorrência otimista.
- 18.12. Como a granularidade dos itens de dados afeta a execução do controle de concorrência? Quais fatores afetam a seleção do tamanho de granularidade para itens de dado?

18.8 Resumo 437

- 18.13. Quais tipos de bloqueios são necessários para inserir e remover operações?
- 18.14. O que é bloqueio de granularidade múltipla? Sob quais circunstâncias ele é usado?
- 18.15. O que são os bloqueios de intenção?
- 18.16. Quando as travas são usadas?
- 18.17. O que é um registro fantasma? Discuta o problema que um registro fantasma pode causar para o controle de concorrência.
- 18.18. Como o bloqueio de índices resolve o problema do fantasma?
- 18.19. O que é um bloqueio de predicado?

Exercícios

- 18.20. Prove que o protocolo de bloqueio básico em duas fases garante a serialização de conflito do plano de execuções.
(*Indicação:* mostre que, se um grafo de serialização para o plano de execução possui um ciclo, então pelo menos uma das transações participantes no plano de execução não obedece ao protocolo de bloqueio de duas fases).
- 18.21. Modifique as estruturas de dados para bloqueios de múltiplos modos e os algoritmos para `read_lock(X)`, `write_lock(X)` e `unlock(X)`, de tal forma que a promoção e o rebaixamento de bloqueios seja possível. (*Indicação:* o bloqueio necessita verificar o(s) id(s) da transação que controla(m) o bloqueio, se tiver algum.)
- 18.22. Prove que o bloqueio estrito em duas fases garante planos de execução estritos.
- 18.23. Prove que os protocolos esperar-morrer e ferir-esperar evitam deadlock e starvation. 18.24- Prove que a espera cuidadosa (cautious waiting) evita deadlock.
- 18.25. Aplique o algoritmo de ordenação por timestamp para os planos de execução das figuras 17.8 b e c e determine se o algoritmo permitirá a execução desses planos.
- 18.26. Repita o Exercício 18.25, mas use o método de ordenação por timestamp multiversão.
- 18.27. Por que o bloqueio em duas fases não é usado como um método de controle de concorrência para índices tais como árvores-B?
- 18.28. A matriz de compatibilidade da Figura 18.8 mostra que os bloqueios IS e IX são compatíveis. Explique por que isso é válido.
- 18.29. O protocolo MGL declara que uma transação T pode desbloquear um nó N apenas se nenhum dos filhos do nó N ainda estiverem bloqueados pela transação T. Mostre que, sem essa condição, o protocolo MGL seria incorreto.

Bibliografia Seleccionada

O protocolo de bloqueio em duas fases e o conceito de bloqueios de predicado foram propostos, primeiro, por Eswaran *et al.* (1976). Bernstein *et al.* (1987), Gray e Reuter (1993) e Papadimitriou (1986) enfocaram o controle de concorrência e recuperação. Kumar (1996) priorizou a execução dos métodos de controle de concorrência. Bloqueio é discutido em Gray *et al.* (1975), Lien e Weinberger (1978), Kedem e Silbershatz (1980) e Korth (1983). Deadlocks e grafos wait-for foram formalizados por Holt (1972), e os esquemas esperar-ferir e ferir-morrer são apresentados em Rosenkrantz *et al.* (1978). Cautious waiting (espera cuidadosa) é discutida em Hsu *et al.* (1992). Helal *et al.* (1993) comparam várias abordagens de bloqueio. Técnicas de controle de concorrência são vistas em Bernstein e Goodman (1980) e Reed (1983). Controle de concorrência otimista é analisado em Kung e Robinson (1981) e em Bassiouni (1988). Papadimitriou e Kanellakis (1979) e Bernstein e Goodman (1983) discutem técnicas de multiversão. Ordenação multiversão por timestamp foi proposta em Reed (1978, 1983), e bloqueio multiversão em duas fases é discutido em Lai e Wilkinson (1984). Um método para granularidades de bloqueio múltiplo foi proposto em Gray *et al.* (1975), e os efeitos de granularidade de bloqueio são analisados em Ries e Stone-braker (1977). Bhargava e Reidl (1988) apresentam uma abordagem para escolher dinamicamente vários métodos de controle de concorrência e recuperação. Métodos de controle de concorrência para índices são apresentados em Lehman e Yao (1981) e em Shasha e Goodman (1988). Um estudo da execução de vários algoritmos de controle de concorrência de árvore B é apresentado em Srinivasan e Carey (1991).

Outro trabalho recente no controle de concorrência inclui controle de concorrência baseado em semântica (Badrinath e Ramamritham, 1992), modelos de transação para atividades de longa duração em execução (Dayal *et al.*, 1991) e gerenciamento de transação em multinível (Hasse e Weikum, 1991).

19

Técnicas de Recuperação de Banco de Dados

Neste capítulo discutiremos algumas das técnicas que podem ser usadas para recuperação (ou restauração) de falhas de banco de dados. Já discutimos, na Seção 17.1.4, as diferentes causas de falhas, como quedas de sistema e erros de transação. Cobrimos também, na Seção 17.2, muitos conceitos que são usados pelos processos de recuperação, tais como *log* (histórico) de sistema e pontos de efetivação.

Começaremos a Seção 19.1 com um esquema dos procedimentos de uma restauração típica e a categorização dos algoritmos de recuperação, depois veremos os diversos conceitos de recuperação, incluindo gravação adiantada de *log* (write-ahead logging), as atualizações in loco *versus* sombra, o processo de desfazer (*rolling back*) e o efeito de uma transação incompleta ou com falha. Na Seção 19.2 apresentaremos as técnicas de recuperação baseadas em *atualização adiada*, também conhecida como técnica NO-UNDO/REDO. Na Seção 19.3 analisaremos as técnicas de recuperação baseadas em atualizações imediatas, que incluem os algoritmos UNDO/REDO e UNDO/NO-REDO. Discutiremos, na Seção 19.4, as técnicas conhecidas como *shadowing* ou paginação *shadow*, que podem ser categorizadas como um algoritmo NO-UNDO/NO-REDO. Um exemplo de um esquema prático de recuperação de SGBD, chamado ARIES, será apresentado na Seção 19.5. A recuperação de diversos bancos de dados será brevemente analisada na Seção 19.6. Finalmente, as técnicas para recuperação de falhas catastróficas serão discutidas na Seção 19.7. Nossa ênfase está na descrição conceitual de abordagens diferentes para recuperação. Para descrições das características de recuperação em sistemas específicos, o leitor deve consultar as notas bibliográficas e o manual do usuário para esses sistemas. Técnicas de recuperação estão, freqüentemente, entrelaçadas aos mecanismos de controle de concorrência. Certas técnicas de recuperação são melhores para determinados métodos de controle de concorrência. Tentaremos ver os conceitos de recuperação independentemente dos mecanismos de controle de concorrência, mas discutiremos as circunstâncias sob as quais um dado mecanismo de recuperação é mais adequado a certo protocolo de controle de concorrência.

19.1 CONCEITOS DE RECUPERAÇÃO

19.1.1 Esquema de Recuperação e Categorização de Algoritmos de Recuperação

A recuperação de transações que falharam significa, em geral, que o banco de dados será *restaurado* para o estado de consistência mais recente, exatamente como antes do momento da falha. Para isso, o sistema deverá manter informação sobre as alterações que foram aplicadas aos itens de dado pelas várias transações. Essa informação é, em geral, armazenada no ***log* (histórico) do sistema**, conforme vimos na Seção 17.2.2. Uma estratégia típica para recuperação pode ser resumizada informalmente como segue:

19.1 Conceitos de Recuperação 439

1. Se houver um dano extenso em uma grande porção do banco de dados, por conta de falha catastrófica, tal como um *crash* de disco, o método de recuperação restaura uma cópia anterior do banco de dados, que estava *guardada* em um arquivo de armazenamento (normalmente uma fita), e o reconstrói num estado mais atual, reaplicando ou *refazendo* as operações das transações *armazenadas* no *log* até o instante da falha (restore de um *backup*).

2. Quando o banco de dados não for danificado fisicamente, mas se tornar inconsistente por causa de uma falha não-catastrófica, dos tipos 1 a 4 da Seção 17.1.4, a estratégia é reverter quaisquer mudanças que causaram a inconsistência *desfazendo* algumas operações. Também será necessário *refazer* algumas operações, de forma a restaurar um estado consistente do banco de dados, conforme veremos. Nesse caso, não necessitamos de uma cópia arquivada completa do banco de dados. De preferência, as entradas mantidas em um *log* on-line do sistema serão consultadas durante a recuperação.

Conceitualmente, podemos distinguir duas técnicas principais de recuperação de falhas não-catastróficas de transação: (1) atualização adiada e (2) atualização imediata. Na técnica de atualização adiada, somente se atualiza o banco de dados fisicamente no disco logo *depois* que uma transação alcance seu ponto de efetivação; as atualizações são, então, gravadas no banco de dados. Antes que a efetivação seja alcançada, todas as transações atualizam seus registros no espaço de transação local (ou *buffers*). Durante a efetivação, as atualizações são, primeiro, persistentemente registradas no *log* e, então, gravadas no banco de dados. Se uma transação falhar antes de alcançar seu ponto de efetivação, ela não terá mudado o banco de dados de forma nenhuma, assim, um UNDO não será necessário. Pode ser necessário um REDO nos efeitos das operações de uma transação que foram efetivadas no *log* porque seus efeitos poderão não ter sido, ainda, registrados no banco de dados. Portanto, a atualização adiada é também conhecida como o algoritmo NO-UNDO/REDO. Discutiremos essa técnica na Seção 19.2.

Nas técnicas de atualização imediata, o banco de dados pode ser atualizado por algumas operações de uma transação *antes* que ela alcance seu ponto de efetivação. Entretanto, essas operações estarão normalmente registradas no *log em disco*, pois uma gravação forçada foi feita *antes* que elas fossem aplicadas ao banco de dados, tornando possível a recuperação. Se uma transação falhar depois de registrar algumas mudanças no banco de dados, mas antes de alcançar seu ponto de efetivação, os efeitos de suas operações no banco de dados deverão ser desfeitos, isto é, a transação deve ser revertida. No caso geral de atualização imediata, ambas, *undo* e *redo*, devem ser exigidas para a recuperação. Essa técnica, conhecida como algoritmo UNDO/REDO, exige ambas as operações e é mais freqüentemente usada na prática. Uma variação do algoritmo, em que todas as atualizações são registradas no banco de dados antes que uma transação seja efetivada, requer apenas *undo*, assim, ela é conhecida como o algoritmo UNDO/NO-REDO. Veremos essas técnicas na Seção 19.3.

19.1.2 Caching (Bufferização) de Blocos de Disco

Em geral, o processo de recuperação está fortemente entrelaçado com as funções do sistema operacional — em particular, com a bufferização e o *caching* de páginas de disco na memória principal. Normalmente, uma ou mais páginas de disco, que incluem os itens de dados a serem atualizados, são escondidas (*cached*) nos *buffers* da memória principal e lá são atualizadas, antes de serem gravadas de volta no disco. O *caching* de páginas em disco é tradicionalmente uma função do sistema operacional, mas, por causa de sua importância para a eficiência dos procedimentos de recuperação, ele é manipulado pelo SGBD por meio de chamadas de rotinas de baixo nível do sistema operacional.

Muitas vezes, é conveniente considerar a recuperação em termos de páginas de disco do banco de dados (blocos). Normalmente, uma coleção de *buffers* em memória, chamada *cache* de SGBD, é mantida sob o controle do SGBD com o propósito de controlar esses *buffers*. É usado um catálogo para o cache para controlar quais itens do banco de dados estão nos *buffers*. Ele pode ser uma tabela de entradas <endereço das páginas do disco, localização do buffer>. Quando o SGBD solicitar uma ação em algum item, ele primeiro verificará o catálogo de *cache* para determinar se alguma das páginas de disco em *cache* contém o item. Se não contiver, então o item deverá ser localizado no disco, e as páginas de disco apropriadas serão copiadas no *cache*. Pode ser que seja necessário **substituir** (ou *flush*, *expelir*) alguns dos *buffers* em *cache* para que haja espaço disponível para o novo item. Algumas estratégias dos sistemas operacionais para substituição de página, tais como a menos usada recentemente (*least recently used* — LRU) ou o primeiro que entra é o primeiro que sai (*first-in-first-out*, FIFO), podem ser usadas na seleção dos *buffers* que serão substituídos. É associado a cada *buffer* no *cache* um bit **sujeira** (*dirty bit*), que pode ser incluído na entrada do catálogo para indicar se o *buffer* foi ou não modificado. Quando uma página é lida pela primeira vez no disco do banco de dados e passa para um *buffer* de *cache*, o catálogo será atualizado com o novo endereço da página, e o bit sujeira é marcado com 0 (zero). Assim que o *buffer* for

1 Similar ao conceito de *tabelas de páginas* usado pelo sistema operacional.

modificado, o bit sujeira correspondente à entrada do catálogo é apontado para 1 (um). Quando o conteúdo do *buffer* for substituído (flushed) no *cache*, ele deverá, primeiro, ser regravado na página de disco correspondente apenas se seu bit sujeira for 1. Outro bit, chamado bit **pin-unpin**, também é necessário — uma página no *cache* está **pinned (fixada)** (valor de bit 1 — um) — se ela não puder escrever de volta no disco.

Duas estratégias podem ser empregadas quando um *buffer* modificado for regravado no disco. A primeira estratégia, conhecida como **atualizando no local (in-place updating)**, regrava o *buffer* no mesmo local original do disco, sobrescrevendo, então, o valor dos itens de dados alterados no disco. Portanto, apenas uma única cópia de cada bloco de disco do banco de dados será mantida. A segunda estratégia, conhecida como **shadowing**, grava um *buffer* atualizado em uma localização diferente do disco, assim, diversas versões dos itens de dados serão mantidas. Em geral, o antigo valor dos itens de dados, antes da atualização, é chamado de **imagem anterior (BFIM)**, e o novo valor, depois da atualização, é chamado de **imagem posterior (AFIM)**. Em **shadowing**, ambas, a BFIM e a AFIM, poderão ser mantidas no disco, portanto, não é estritamente necessário manter um *log* para recuperação. Discutiremos brevemente, na Seção 19.4, a recuperação baseada em **shadowing**.

19.1.3 Registro Adiantado em Log, Roubado/Não-Roubado e Forçado/Não-Forçado

Quando um 'atualizando no local' for usado, será necessário usar o *log* para recuperação (Seção 17.2.2). Nesse caso, o mecanismo de recuperação deve garantir que a BFIM (imagem anterior) do item de dados seja registrada em uma entrada de *log* apropriada, e que essa entrada de *log* seja transferida (*flushed*) para o disco, antes que a BFIM seja sobrescrita pela AFIM (imagem posterior) no banco de dados do disco. Em geral, esse processo é conhecido como **registro adiantado em log**. Antes que possamos descrever um protocolo para o registro adiantado em *log*, necessitamos distinguir entre dois tipos de informação de entrada em *log* para um comando de escrita (*write*): (1) as informações necessárias para UNDO e (2) aquelas necessárias para REDO. Uma **entrada de log do tipo REDO** inclui o **novo valor (AFIM)** do item gravado pela operação, uma vez que será necessário *refazer* o efeito da operação pelo *log* (apontando o valor do item no banco de dados para seu AFIM). As **entradas de log de tipo UNDO** incluem o **valor antigo (BFIM)** do item, uma vez que é necessário *desfazer* o efeito da operação pelo *log* (apontando o valor do item no banco de dados de volta à sua BFIM). Em um algoritmo UNDO/REDO, ambos os tipos de entradas de *log* são combinados. Além disso, quando houver possibilidade de ocorrer retorno em cascata, entradas *ler_item* (ler item) no *log* serão consideradas entradas de tipo UNDO (Seção 19.1.5).

Como mencionado, o *cache* do SGBD controla os blocos de disco escondidos do banco de dados, que incluem não apenas *blocos de dados*, mas também os *blocos de índice* e os *blocos de log* do disco. Quando um registro de *log* for gravado, ele é armazenado no bloco de *log* corrente, dentro do *cache* do SGBD. O *log* é simplesmente um arquivo de disco seqüencial (*append-only*) e o *cache* do SGBD poderá conter diversos blocos de *log* (por exemplo, os *n* últimos blocos de *log*) que serão gravados no disco. Quando uma atualização em um bloco de dados — armazenado no *cache* do SGBD — for feita, um registro de *log* associado será gravado no último bloco de *log* no *cache* do SGBD. Com a abordagem registro adiantado no *log*, os blocos de *log* que contiverem registros de *log* associados a uma atualização em um bloco de dados em particular deverão ser gravados primeiro no disco, antes que o bloco de dados possa, ele mesmo, ser regravado no disco.

A terminologia padrão para recuperação de SGBD inclui os termos **roubado/não-roubado (steal/no-steal)** e **forçado/não-forçado (force/no-force)**, que especifica quando uma página do banco de dados poderá ser gravada em disco a partir do *cache*:

1. Se uma página em *cache* atualizada por uma transação *não puder* ser gravada antes que a transação se efetive, ela será chamada de **abordagem não-roubada (no-steal)**. O bit de *pin-unpin* indicará se a página não puder ser escrita de volta no disco. Do contrário, se o protocolo permitir o *buffer* atualizado *antes* que a transação se efetive, ele será chamado **roubado (steal)**. *Steal* será usado quando o gerenciador do *cache (buffer)* do SGBD necessitar de um *frame* de *buffer* para outra transação, e o gerenciador do *buffer* substituir uma página existente que tenha sido atualizada, mas cuja transação não tenha se efetivado.
2. Se todas as páginas atualizadas por uma transação forem imediatamente escritas no disco quando a transação se efetivar, ela será chamada **abordagem forçada (force)**. Do contrário, será chamada **não-forçada (no-force)**. O esquema de recuperação de atualização adiado, da Seção 19.2, segue uma abordagem *no-steal*. Entretanto, os sistemas de banco de dados normalmente empregam uma estratégia *roubada/não-forçada (steal/no-force)*. A vantagem da *steal* é que ela evita a necessidade de um espaço muito grande de *buffer* para o armazenamento em memória de todas as páginas atualizadas. A vantagem do *no-force* é que uma página atualizada por uma transação efetivada deverá ainda estar no *buffer* quando outra

- 2 Na prática, a atualização no local é usada na maioria dos sistemas.

1 9.1 Conceitos de Recuperação 441

transação necessitar de atualização, eliminando, assim, o custo de I/O para ler novamente essa página no disco. Isso pode oferecer uma economia substancial no número de operações de I/O quando uma dada página for atualizada pesadamente por diversas transações.

Para permitir a recuperação quando uma atualização no local (*in-place*) for usada, as entradas apropriadas exigidas para a recuperação devem ser permanentemente registradas no disco de *logon*, antes que as alterações sejam aplicadas no banco de dados. Por exemplo, considere o seguinte protocolo de registro adiantado em *log* (*write-ahead logging* — WAL), para um algoritmo de recuperação que exige ambas, UNDO e REDO:

1. A imagem anterior de um item não pode ser sobrescrita por sua imagem posterior no banco de dados do disco; até que se registre todo *log* de tipo UNDO para transações atualizadas — acima desse ponto —, seria uma gravação forçada (*force-written*) no disco.
2. A operação de efetivação (*commit*) de uma transação não poderá ser completada até que se registre todo *log* de tipo REDO e UNDO para que essa transação seja de gravação forçada (*force-written*) no disco.

Para facilitar o processo de recuperação, o subsistema de recuperação deverá manter algumas listas relacionadas às transações processadas no sistema. Estas incluem uma lista das transações ativas que iniciaram, mas que ainda não foram efetivadas, e poderá incluir também as listas de todas as transações efetivadas e abortadas, desde o último *checkpoint* (ponto de controle — veja próxima seção). A manutenção dessas listas torna o processo de recuperação mais eficiente.

19.1.4 Checkpoints no Log de Sistema e Fuzzy Checkpointing

Outro tipo de entrada no *log* é chamado de *checkpoint*. Um registro [*checkpoint*] é escrito periodicamente dentro do *log*, no ponto em que o sistema grava no banco de dados no disco todos os *buffers* do SGBD que tiverem sido modificados. Como consequência, todas as transações que tiverem suas entradas [*commit*, T] no *log*, antes de uma entrada [*checkpoint*], não necessitarão ter suas operações WRITE refeitas no caso de queda do sistema, uma vez que todas as suas atualizações foram registradas no banco de dados em disco, durante o *checkpointing*.

O gerenciador de recuperação de um SGBD deve decidir em quais intervalos submeter um *checkpoint*. O intervalo deve ser medido em tempo — quer dizer, a cada *m* minutos — ou no número *t* de transações efetivadas desde o último *checkpoint*, e os valores de *m* ou *t* são parâmetros do sistema. Submeter um *checkpoint* consiste nas seguintes ações:

1. Suspender a execução das transações temporariamente.
2. Forçar a gravação no disco de todos os *buffers* na memória principal que tenham sido alterados.
3. Escrever um registro de [*checkpoint*] no *log* e forçar a gravação do *log* no disco.
4. Reassumir a execução das transações.

Como consequência do passo 2, um registro de *checkpoint* no *log* pode incluir também informações adicionais, como uma lista dos *ids* (identificações) das transações ativas e as localizações (endereços) de todos os registros, do mais antigo ao mais recente (último) no *log*, de cada transação ativa. Isso pode facilitar o 'desfazer' das operações da transação, no caso de ser necessária sua reversão.

O tempo necessário para forçar a gravação de todos os *buffers* de memória modificados pode atrasar o processamento da transação por causa do passo 1. Para reduzir esse atraso, na prática, é comum o uso de uma técnica chamada *fuzzy checkpointing*. Nessa técnica, o sistema poderá reassumir o processamento das transações depois que o registro [*checkpoint*] for escrito no *log*, sem ter de esperar o passo 2 terminar. Entretanto, até que o passo 2 esteja completo, o registro [*checkpoint*] anterior permanecerá válido. Para fazer isso, o sistema manterá um ponteiro para o *checkpoint* válido, que continuará a apontar para o registro [*checkpoint*] no *log*.

Uma vez concluído o passo 2, o ponteiro será mudado para o ponto do novo *checkpoint* no *log*.

19.1.5 Reverter (Rollback) uma Transação

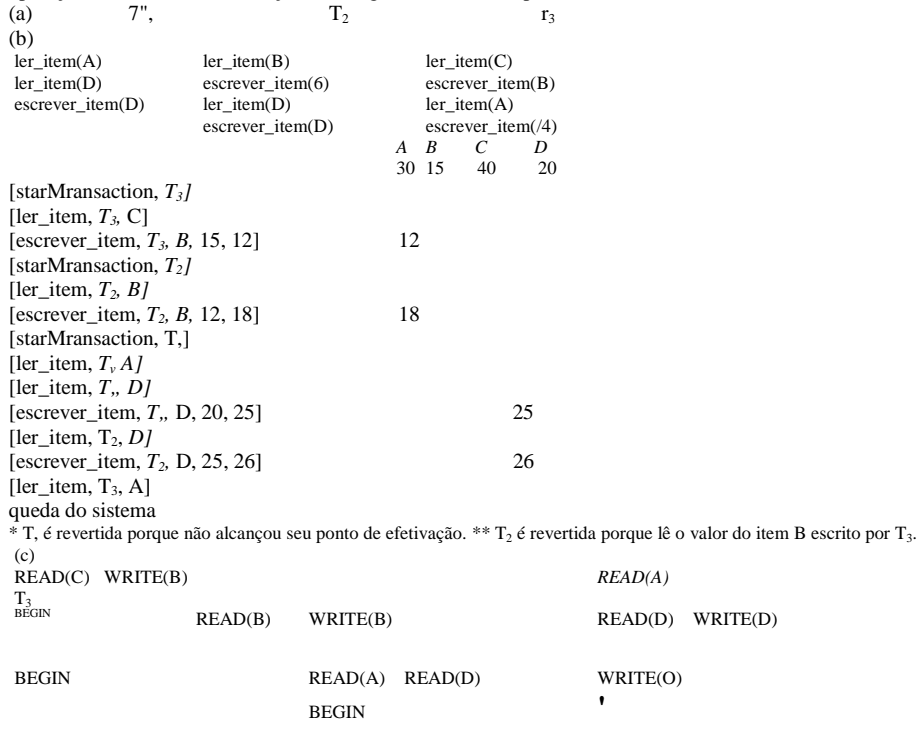
Se uma transação falhar por qualquer razão, depois de atualizar o banco de dados, pode ser necessário reverter (*rollback*) a transação. Se algum valor de item de dados tiver sido alterado pela transação e gravado no banco de dados, ele deverá ser restaurado aos seus valores anteriores (BFIMs). As entradas de *log* do tipo desfazer (UNDO) são usadas para restaurar os valores antigos dos itens de dado que deverão ser revertidos.

3 O termo *checkpoint* (ponto de controle) tem sido usado para descrever situações restritivas em alguns sistemas, como no DB2. Também tem sido usado na literatura para descrever conceitos inteiramente diferentes.

442 Capítulo 19 Técnicas de Recuperação de Banco de Dados

Se uma transação T for revertida, qualquer transação S que tenha, nesse ínterim, lido o valor de algum item de dado ? escrito por T também deverá ser revertida. Similarmente, uma vez que S seja revertida, qualquer transação R que tenha lido o valor de algum item de dado Y escrito por S também deverá ser revertida; e assim por diante. Esse fenômeno é chamado **reversão em cascata** (*cascading rollback*) e pode ocorrer quando o protocolo de recuperação garantir planos *recuperáveis*, mas não garantir planos restritos ou *livres de cascata* (*cascadeless*, Seção 17.4.2). A reversão em cascata, compreensivelmente, pode ser mais complexa e consumir mais tempo. Por isso, quase todos os mecanismos de recuperação são projetados de modo que a reversão em cascata *nunca* seja exigida.

A Figura 19.1 mostra um exemplo no qual a reversão em cascata é exigida. As operações de leitura e gravação de três transações são mostradas na Figura 19.1a. A Figura 19.1b mostra o log de sistema no ponto de queda de um dado plano de execução dessas transações. Os valores dos itens de dado A , B , C e D , que são usados pelas transações, são mostrados à direita das entradas do log do sistema. Adotamos como valores originais dos itens, mostrados na primeira linha, $A = 30, B = 15, C = 40$ e $D = 20$. No ponto de falha do sistema, a transação T_3 não alcançou sua conclusão e deverá ser revertida. As operações **WRITE** de T_3 , marcadas por um * na Figura 19.1b, são as operações de T_3 que serão desfeitas durante a reversão da transação. A Figura 19.1c mostra graficamente as operações das diferentes transações ao longo do eixo do tempo.



queda do sistema -

FIGURA 19.1 Ilustrando a reversão (*rollback*) em cascata (um processo que nunca ocorre em planos restritos ou livres de cascata — *cascadeless*). (a) As operações *read*(ler) e *write*(gravar) das três transações, (b) *Log*(histórico) do sistema no ponto da queda (*crash*). (c) Operações antes da queda.

19.2 Técnicas de Recuperação Baseadas na Atualização Adiada 443

Devemos, agora, verificar a reversão em cascata. Na Figura 19.1c vemos que a transação T_2 lê o valor do item B que foi escrito pela transação T_3 ; isso também pode ser determinado examinando-se o *log*. Como T_1 será revertida, T_2 deverá ser revertida também. As operações WRITE de T_2 , marcadas por ** no *log*, são as únicas não desfeitas. Observe que as operações escrever_item precisarão ser desfeitas durante a reversão da transação; operações ler_item são registradas no *log* apenas para determinar se a reversão em cascata das transações adicionais será necessária.

Na prática, a reversão em cascata de transações *nunca* é exigida porque os métodos práticos de recuperação garantem planos livres de cascata ou restritos. Portanto, também não é necessário registrar as operações ler_item no *log*, porque elas são necessárias apenas para estabelecer a reversão em cascata.

19.2 TÉCNICAS DE RECUPERAÇÃO BASEADAS NA ATUALIZAÇÃO ADIADA

A idéia contida nas técnicas de atualização adiada é postergar qualquer atualização real no banco de dados até que a transação complete sua execução com sucesso e alcance seu ponto de efetivação. Durante a execução da transação, as atualizações são registradas apenas no *log* e nos *buffers* de *cache*. Quando a transação alcançar seu ponto de efetivação e se forçar à gravação do *log* no disco, as atualizações serão registradas no banco de dados. Se uma transação falhar antes de alcançar seu ponto de efetivação, não será necessário desfazer nenhuma operação, porque a transação não afetou o banco de dados no disco, de forma nenhuma. Embora isso simplifique a recuperação, não poderá ser usado na prática, a menos que as transações sejam curtas e cada transação altere poucos itens. Para outros tipos de transações, há execução potencial fora do espaço de buffer, dado que as mudanças da transação precisam ser garantidas nos *buffers* de *cache* até o ponto de efetivação. Podemos declarar um protocolo típico de atualização adiada como segue:

1. Uma transação não pode mudar o banco de dados em disco até que ela alcance seu ponto de efetivação.
2. Uma transação não alcança seu ponto de efetivação até que todas as suas operações de atualização sejam registradas no *log* e até que seja forçada a gravação do *log* no disco.

Observe que o passo 2 é uma reafirmação do protocolo de registro adiantado no *log* (*write-ahead logging* — WAL). Como o banco de dados nunca é atualizado no disco até que a transação seja efetivada, nunca será necessária qualquer operação de UNDO (desfazer). Este é conhecido como **algoritmo de recuperação de NO-UNDO/REDO**. REDO (refazer) será necessário no caso de o sistema falhar depois que uma transação for efetivada, mas antes que todas as suas mudanças sejam registradas no banco de dados em disco. Nesse caso, as operações da transação serão refeitas a partir das entradas de *log*.

Geralmente, o método de recuperação de falhas está intimamente relacionado ao método de controle de concorrência em sistemas multiusuários. Discutiremos, primeiro, a recuperação em sistemas monousuários, em que o controle de concorrência não é necessário, tanto que podemos entender o processo de recuperação independentemente de qualquer método de controle de concorrência. Depois, veremos como o controle de concorrência pode afetar o processo de recuperação.

19.2.1 Recuperação Usando Atualização Adiada em um Ambiente Monousuário

Em tais ambientes, o algoritmo de recuperação pode ser bastante simples. O algoritmo RDU_S (*Recovery Using Deferred Update in a Single-user environment* — recuperação usando atualização adiada em ambiente monousuário) usa o procedimento REDO, apresentado a seguir, para refazer certas operações escrever_item:

PROCEDIMENTO RDUS — Usa duas listas de transações: as transações efetivadas desde o último *checkpoint* e as transações ativas (pelo menos uma transação irá cair nessa categoria porque o sistema é monousuário). Aplicar a operação REDO para todas as operações escrever_item das transações efetivadas do *log*, seguindo a seqüência em que elas foram escritas no *log*. Reiniciar as transações ativas.

O procedimento REDO é definido como segue:

REDO(WRITE_OP) — Refazer uma operação escrever_item em WRITE_OP consiste em examinar sua entrada no *log* [escrever_item, T, X, novo_valor] e apontar o valor do item X no banco de dados para novo_valor, que é a imagem posterior (AFIM).

- 4 A atualização adiada pode, normalmente, ser caracterizada como uma *abordagem não-roubada* (*no-steal*).

444 Capítulo 19 Técnicas de Recuperação de Banco de Dados

A operação REDO deve ser **equivalente em potência (*idempotent*)** — isto é, executá-la várias vezes é equivalente a executá-la apenas uma vez. De fato, todo o processo de recuperação deve ser equivalente em potência. Isso porque, se o sistema falhar durante o processo de recuperação, a próxima tentativa de recuperação poderia ter de REDO (refazer) certas operações escrever_item que já tivessem sido refeitas durante o primeiro processo de recuperação. O resultado da recuperação de uma queda do sistema que ocorresse *durante a recuperação* seria o mesmo que o resultado da recuperação *quando não há queda durante a recuperação!* Observe que apenas as transações da lista ativa não terão afetado o banco de dados por causa do protocolo de atualização adiada, e elas serão totalmente ignoradas pelo processo de recuperação porque nenhuma de suas operações foram refletidas no banco de dados em disco. Entretanto, essa transação deverá, agora, ser reiniciada automaticamente pelo processo de recuperação ou manualmente pelo usuário.

A Figura 19.2 mostra um exemplo de recuperação em um ambiente monousuário, no qual a primeira falha ocorre durante a execução da transação T_2 , conforme mostrado na Figura 19.2b. O processo de recuperação irá refazer a entrada [escrever_item, T_1 , D, 20] do log, apontando o valor do item D para 20 (seu novo valor). As entradas [write, T_2 , ...] no log são ignoradas pelo processo de recuperação porque T_2 não foi efetivada. Se uma segunda falha ocorrer durante a recuperação da primeira, o mesmo processo de recuperação será repetido do início ao fim, com resultados idênticos.

(a) T_1 T_2
 ler_item(4) ler_item(B)
 ler_item(D) escrever_item(B)
 escrever_item(D) ler_item(D)
 escrever_item(D)

(b)
 [inicia_transacao, 7",]
 [ler_item, T_1 , D, 20]
 [confirma, T_1]
 [inicia_transacao, 7" T_2]
 [ler_item, T_2 , B, 10]
 [escrever_item, T_2 , D, 25] <- queda do sistema
 As operações [escrever_item, ...] de 7, são refeitas.

As entradas no log de T_2 são ignoradas pelo processo de recuperação.

FIGURA 19.2 Um exemplo de recuperação usando atualização em ambiente monousuário. (a) As operações READ e WRITE de duas transações, (b) O log do sistema no instante da queda.

19.2.2 Atualização Adiada com Execução Concorrente em um Ambiente Multiusuário

Para sistema multiusuários com controle de concorrência, o processo de recuperação pode ser mais complexo, dependendo dos protocolos usados para o controle de concorrência. Em muitos casos, os processos de controle de concorrência e de recuperação são inter-relacionados. Em geral, quanto maior o grau de concorrência desejado, maior o tempo consumido para a recuperação.

Considere um sistema em que o controle de concorrência usa bloqueio estrito em duas fases, assim, os bloqueios nos itens permanecerão em efeito *até que a transação alcance seu ponto de efetivação*. Depois disso, os bloqueios serão liberados, garantindo planos restritos e serializáveis. Supondo que as entradas de [checkpoint] foram feitas no log, um possível algoritmo de recuperação para esse caso, que chamamos RDU_M (Recovery using Deferred Update in a Multiuser environment — recuperação usando atualização adiada em ambiente multiusuário) é dado a seguir. Esse procedimento usa o procedimento REDO definido anteriormente.

PROCEDIMENTO RDU_M (COM CHECKPOINTS) - Usa duas listas de transações mantidas pelo sistema: as transações T efetivadas desde o último checkpoint (**lista de efetivação**) e as transações T' ativas (**lista de ativas**). REDO (refazer) todas as operações WRITE das transações efetivadas no log, *na sequência em que elas se encontram no log*. As transações que estiverem ativas, mas não efetivadas, serão efetivamente canceladas e deverão ser novamente submetidas.

A Figura 19.3 mostra um plano de execução possível das transações. Quando foi feito o *checkpoint* no tempo t_1 , a transação T_1 foi efetivada, enquanto as transações T_3 e T_4 não o foram. Antes da queda do sistema no tempo t_2 , T_3 e T_2 foram efetivadas,

19.2 Técnicas de Recuperação Baseadas na Atualização Adiada 445

mas T_4 e T_5 não. De acordo com o método RDU_M, não é necessário refazer as operações de escrever_item da transação T_1 , — ou qualquer transação efetivada antes do tempo t_1 do último *checkpoint*. Entretanto, as operações escrever_item de T_2 e T_3 devem ser refeitas porque ambas as transações alcançaram seus pontos de efetivação depois do último *checkpoint*. Lembremos que a gravação no *log* é forçada antes da efetivação de uma transação. As transações T_4 e T_5 são ignoradas: elas são efetivamente canceladas ou revertidas porque nenhuma de suas operações write_item foram registradas no banco de dados sob o protocolo de atualização adiada. Iremos nos referir à Figura 19.3 depois de ilustrarmos outros protocolos de recuperação.

²2 | Tempo
checkpoint-----' queda do sistema -
FIGURA 19.3 Um exemplo de recuperação em ambiente multiusuário.

Podemos tornar o algoritmo de recuperação NO-UNDO/REDO *mais eficiente* notando que, se um item de dado X — como indicado nas entradas de *log* — tiver sido atualizado por mais de uma transação efetivada desde o último *checkpoint*, apenas será necessário REDO a última atualização de X do *log* durante a recuperação. De qualquer forma, as outras atualizações seriam sobrescritas por esse último REDO. Nesse caso, iniciamos pelo fim do *log*, assim, sempre que um item for refeito, ele será adicionado a uma lista de itens refeitos. Antes de o REDO ser aplicado a um item, a lista é verificada; se o item aparecer na lista, ele não será refeito novamente, uma vez que o último valor já foi recuperado.

Se uma transação for interrompida por qualquer razão (digamos, pelo método de detecção de impasse — *deadlock*), ela simplesmente será resubmetida, uma vez que não alterou o banco de dados em disco. Uma desvantagem do método descrito aqui é que ele limita a concorrência de transações porque *todos os itens permanecem bloqueados até que a transação alcance seu ponto de efetivação*. Além disso, ele pode exigir espaço excessivo de *buffer* para controlar todos os itens atualizados, até que as transações sejam efetivadas. O benefício principal do método é que as operações da transação *nunca terão de ser desfeitas* por duas razões:

1. Uma transação não registrará nenhuma mudança no banco de dados em disco até imediatamente após seu ponto de efetivação — isto é, até que ela complete sua execução com sucesso. Portanto, uma transação nunca será revertida em função de falhas durante a execução de transações.

2. Uma transação nunca lerá o valor de um item escrito por uma transação ainda não-efetivada, porque seus itens permanecerão bloqueados até que a transação alcance seu ponto de efetivação. Portanto, não ocorrerá reversão em cascata.

A Figura 19.4 mostra um exemplo de recuperação para um sistema multiusuário que utiliza os métodos de recuperação e controle de concorrência descritos.

19.2.3 Ações de Transação que Não Afetam o Banco de Dados

Em geral, há ações que *não* afetam o banco de dados, tais como gerar e imprimir mensagens ou relatórios de informações recuperadas do banco de dados. Se uma transação falhar antes de se completar, podemos não desejar que o usuário obtenha esses relatórios. Se relatórios errados forem produzidos, parte do processo de recuperação teria de informar ao usuário que eles estão errados, uma vez que o usuário pode tomar uma ação com base nesses relatórios que afetem o banco de dados. Portanto, tais relatórios deveriam ser gerados somente *depois que a transação alcançasse seu ponto de efetivação*. Um método comum para o tratamento de tais ações é escolher comandos que gerem relatórios, mas os mantenham como tarefas em lote (*batch jobs*), executadas apenas depois que a transação alcançar seu ponto de efetivação. Se a transação falhar, as tarefas em lote serão canceladas.

446 Capítulo 19 Técnicas de Recuperação de Banco de Dados

(a) ler_item(/A) ler_item(D) escrever_item(D)
 (b) [inicia_transacao, 7",.] [escrever_item, T,, D, 20] [confirma, 7",.] [checkpoint] [inicia_transacao, 7"J [escrever_item, T₄, S, 15] [escrever_item, T₁, B, 20] [confirma, 7J [inicia_transacao, T₂] [escrever_item, T₂, 0, 12] [inicia_transacao, T₃] [escrever_item, T₃, A, 30] [escrever_item, 7",, D, 25] <
 ler_item(S) escrever_item(S) ler_item(D) escrever_item(D)
 ler_item(A) escrever_item(A) ler_item(C) escrever_item(C)
 7.
 ler_item(S) escrever_item(S) ler_item(/4) escrever_item(A)
 queda do sistema

T₂ e T₃ são ignoradas porque não alcançaram seus pontos de efetivação.

7"4 é refeita porque seu ponto de efetivação está depois do último *checkpoint* do sistema.

FIGURA 19.4 Um exemplo de recuperação que usa atualização adiada com transações concorrentes, (a) As operações REAO e WRITE de quatro transações, (b) Log do sistema no ponto de queda.

19.3 TÉCNICAS DE RECUPERAÇÃO BASEADAS EM ATUALIZAÇÃO IMEDIATA

Nessas técnicas, quando uma transação emite um comando de atualização, o banco de dados pode ser 'imediatamente' atualizado, sem nenhuma necessidade de esperar que a transação alcance seu ponto de efetivação. Entretanto, uma operação de atualização deve, ainda assim, ser registrada no *log* (em disco) *antes* de sua aplicação no banco de dados — usando o protocolo registro adiantado em log (*write-ahead logging*) —, de tal forma que possamos recuperá-la em caso de falha.

Deve haver condições para *desfazer* o efeito de operações de atualização que tenham sido aplicadas ao banco de dados por uma *transação falha*. Isso é feito revertendo a transação e desfazendo os efeitos de suas operações *escrever_item*. Teoricamente, podemos identificar dois tipos principais de categorias de algoritmos de atualização imediata. Se as técnicas de recuperação garantirem que todas as atualizações de uma transação sejam registradas no banco de dados em disco antes *de a transação se efetivar*, nunca haverá necessidade de REFAZER quaisquer operações das transações efetivadas. Este é chamado algoritmo de recuperação UNDO/NO-REDO. Por outro lado, se for permitido à transação se efetivar antes que todas as suas mudanças sejam escritas no banco de dados, temos o caso mais geral, conhecido por **algoritmo** de recuperação UNDO/REDO. Essa é, também, a técnica mais complexa. A seguir, discutiremos dois exemplos de algoritmos UNDO/REDO e os deixaremos como um exercício para o leitor desenvolver a variação de UNDO/NO-REDO. Na Seção 19.5 descreveremos uma abordagem mais prática, conhecida como técnica de recuperação ARIES.

19.3.1 Recuperação UNDO/REDO Baseada em Atualização Imediata em um Ambiente Monousuário

Se ocorrer uma falha num sistema monousuário, a transação em execução (ativa) no momento da falha pode ter registrado algumas mudanças no banco de dados. O efeito de todas essas operações deverá ser desfeito. O algoritmo de recuperação RIU_S (*Recovery using Immediate Update in a Single-user environment* — recuperação usando atualização imediata em um ambiente monousuário) usa o procedimento REDO, definido anteriormente, e também o procedimento UNDO, definido abaixo.

PROCEDIMENTO RIU_S

1. Usar duas listas de transações mantidas pelo sistema: as transações efetivadas desde o último *checkpoint* e as transações ativas (pelo menos uma transação dessa categoria irá falhar, pois o sistema é monousuário).
2. Desfazer todas as operações *escrever_item* da transação *ativa* do *log*, usando o procedimento UNDO descrito a seguir.

19.4 Paginação *Shadow* (Sombra) 447

3. Refazer as operações escrever_item das transações *efetivadas* do *log*, na seqüência em que elas foram gravadas no log, usando o procedimento REDO descrito anteriormente.

O procedimento UNDO é definido como segue:

UNDO (WRITE_OP): Desfazer uma operação escrever_item write_op consiste em examinar suas entradas de log [escrever_item, T, X, valor_antigo, valor_novo] e apontar o valor do item X no banco de dados para o valor antigo, que pertence à imagem anterior (BFIM). Para desfazer um número de operações escrever_item de uma ou mais transações do *log*, deve-se manter a *ordem inversa* da ordem na qual as operações foram gravadas no log.

19.3.2 Recuperação UNDO/REDO Baseada em Atualização Imediata com Execução Concorrente

Quando a execução concorrente for permitida, o processo de recuperação dependerá novamente dos protocolos usados para o controle de concorrência. O procedimento RIU_M (*Recovery using Immediate Updates for a Multiuser environment* — recuperação usando atualizações imediatas para um ambiente multiusuário) esboça um algoritmo de recuperação para transações concorrentes com atualização imediata. Assumimos que o *log* inclui *checkpoints* e que o protocolo de controle de concorrência produz *planos restritos* — como, por exemplo, o protocolo de bloqueio em duas fases estrito. Lembremos que um plano restrito não permite que uma transação leia ou grave um item, a menos que a transação que gravou o item por último tenha sido efetivada (ou abortada e revertida). Entretanto, deadlocks podem ocorrer no bloqueio em duas fases estrito, exigindo, assim, abortar e UNDO (desfazer) as transações. Para um plano restrito, UNDO (desfazer) uma operação implica retornar o antigo valor ao item (BFIM).

PROCEDIMENTO RIUM

1. Usar duas listas de transações mantidas pelo sistema: a das transações efetivadas desde o último *checkpoint* e a das transações ativas.
2. Desfazer todas as operações escrever_item feitas pelas transações ativas(não-efetivadas), usando o procedimento UNDO. As operações deverão ser desfeitas na ordem inversa em que elas foram gravadas no *log*.
3. Refazer todas as operações escrever_item das transações *efetivadas* do *log* na seqüência em que elas foram escritas dentro do *log*.

Conforme vimos na Seção 19.2.2, o passo 3 será mais eficiente se for iniciado no *fim do log* e refizer apenas a *última atualização de cada item X*. Sempre que um item for refeito, ele será adicionado à lista de itens refeitos, portanto, não será refeito novamente. Um procedimento similar pode ser planejado para melhorar a eficiência do passo 2.

19.4 PAGINAÇÃO *SHADOW* (SOMBRA)

Este esquema de recuperação não necessita de um *log* quando em ambiente monousuário. Em um ambiente multiusuário, um *log* pode ser necessário para o método de controle de concorrência. A paginação *shadow* considera que o banco de dados é composto por um número de páginas de tamanho fixo (ou blocos de disco) — digamos, n — para propósito de recuperação. Um catálogo com n entradas é construído, no qual a i^{ma} entrada aponta para a i^{sa} página do banco de dados em disco. Se não for muito grande, o catálogo será mantido na memória principal, e todas as referências — de leitura ou gravação — para as páginas do banco de dados passarão por ele. Quando se inicia uma transação, o catálogo corrente — cujas entradas apontam para as mais recentes ou correntes páginas em disco — é copiado em um catálogo *shadow*. O catálogo *shadow* é, então, salvo no disco, enquanto o catálogo corrente é usado pela transação.

Durante a execução da transação, o catálogo *shadow* nunca é modificado. Quando uma operação escrever_item for executada, uma nova cópia da página modificada do banco de dados será criada, mas a cópia antiga dessa página não será sobrescrita. Ao contrário, a nova página será escrita em outro lugar — em algum bloco de disco ainda não usado. A entrada do catálogo corrente será modificada para apontar o novo bloco de disco, enquanto o catálogo *shadow* não será modificado e continuará apontando para o bloco de disco antigo não modificado. A Figura 19.5 ilustra os conceitos de catálogos *shadow* e corrente. Para páginas atualizadas pela transação, serão mantidas duas versões. A versão antiga é referida pelo catálogo *shadow*; a nova versão, pelo catálogo corrente.

5 O catálogo é similar à **tabela de páginas** mantida pelo sistema operacional para cada processo.

448 Capítulo 19 Técnicas de Recuperação de Banco de Dados
 blocos de disco do banco de dados (páginas)
 catálogo corrente (depois de atual as páginas 2,
 catálogo *shadow* (não-atualizado)

FIGURA 19.5 Um exemplo de paginação *shadow*.

Para se recuperar de uma falha durante a execução de uma transação, será suficiente livrar-se das páginas modificadas do banco de dados e descartar o catálogo corrente. O estado do banco de dados antes da execução da transação está totalmente disponível no catálogo *shadow*, e esse estado será recuperado pela reinstalação do catálogo *shadow*. O banco de dados retornará, assim, ao estado anterior ao da transação que estava executando quando ocorreu a falha, e quaisquer páginas modificadas são descartadas. Efetivar uma transação corresponde a descartar o catálogo *shadow* anterior. Uma vez que a recuperação não envolve desfazer nem refazer itens de dado, essa técnica pode ser categorizada como uma técnica NO-UNDO/NO-REDO para recuperação.

Em um ambiente multiusuário com transações concorrentes, os *logs* e os *checkpoints* devem ser incorporados às técnicas de paginação *shadow*. Uma desvantagem da paginação *shadow* é que as páginas em atualização do banco de dados mudam de localização no disco, tornando difícil manter páginas relacionadas juntas no disco sem dispor de estratégias complexas para o gerenciamento de armazenamento. Além disso, se o catálogo for grande, o tempo despendido pelo sistema operacional (*overhead*) para gravar catálogos *shadow* no disco, quando as transações forem efetivadas, é significativo. Uma complicação adicional é como manipular o **lixo** quando uma transação for efetivada. As páginas com referências antigas no catálogo *shadow* que tenham sido atualizadas devem ser liberadas e adicionadas a uma lista de páginas liberadas, para uso posterior. Essas páginas não serão mais necessárias depois que a transação for efetivada. Outra questão é que uma operação, para migrar entre os catálogos corrente e *shadow*, deverá ser implementada como operação atômica.

19.5 O ALGORITMO DE RECUPERAÇÃO ARIES

Descreveremos, agora, o algoritmo ARIES como um exemplo de algoritmo de recuperação usado em sistemas de banco de dados.

ARIES usa uma abordagem roubada/não-forçada (*steal/no-force*) para gravação e está baseado em três conceitos: (1) registro adiantado em log (*write-ahead logging*), (2) repetição de histórico durante o refazer e (3) mudanças do *log* durante o desfazer. Já discutimos o registro adiantado no logna Seção 19.1.3. O segundo conceito, **repetição de histórico**, significa que o ARIES relê todas as ações tomadas pelo sistema de banco de dados antes da queda para reconstruir seu estado *quando a queda ocorreu*.

Transações que não foram efetivadas em tempo de queda (transações ativas) são desfeitas. O terceiro conceito, **usando log durante desfazer**, evitará que o ARIES torne a desfazer operações já desfeitas, caso ocorra uma falha durante a recuperação, com conseqüente reinício do processo de recuperação.

19.5 O Algoritmo de Recuperação ARIES 449

O procedimento de recuperação ARIES consiste em três passos principais: (1) análise, (2) REDO e (3) UNDO. O **passo de análise** identifica as páginas lixo (atualizadas) no buffer e o conjunto das transações ativas no momento da queda. O ponto apropriado no *log*, onde a operação REDO deve começar, também será determinado. É na fase REDO que as atualizações são realmente reaplicadas no banco de dados a partir do *log*. Geralmente, a operação REDO é aplicada apenas às transações efetivadas. Entretanto, no ARIES, esse não é o caso. Certas informações no *log* do ARIES fornecerão o ponto de início para REDO, a partir do qual o REDO deverá ser aplicado às operações até que seja alcançado o fim do *log*. Além disso, informações armazenadas pelo ARIES e pelas páginas de dados permitirão ao ARIES determinar se a operação a ser refeita foi, de fato, aplicada ao banco de dados, assim, não precisará ser reaplicada. Logo, *apenas as operações REDO realmente necessárias* serão aplicadas durante a recuperação. Finalmente, durante a fase UNDO, o *log* será seguido de trás para a frente, e as operações das transações que foram ativadas em tempo de queda serão desfeitas na sequência contrária. As informações necessárias para que o ARIES execute seu procedimento de recuperação compreendem o *log*, a Tabela de Transações e a Tabela de Páginas Lixo. Além disso, é usado o *checkpoint*. Essas duas tabelas são mantidas pelo gerenciador de transações e gravadas no *log* durante o *checkpoint*.

No ARIES, cada registro de *log* tem um **número de sequência de log** (LSN) associado, que é incrementado repetidamente e indica o endereço do registro de *log* no disco. Cada LSN corresponde a uma *mudança específica* (ação) feita por alguma transação. Além disso, cada página de dado armazenará o LSN do *último registro de log correspondente a uma mudança naquela página*. Um registro de *log* é escrito para as seguintes ações: atualização de uma página (*write*), efetivação de uma transação (*commit*), interrupção de uma transação (*abort*), desfazimento de uma atualização (*undo*) e finalização de uma transação (*end*). A necessidade de incluir as primeiras três ações no *log* tem sido discutida, mas as duas últimas precisam de alguma explicação. Quando uma atualização for desfeita, um *registro de compensação* será escrito no *log*. Quando uma transação for encerrada, seja por efetivação ou por interrupção, um *registro de término* será escrito no *log*.

Campos comuns em todos os tipos de *log* incluem: (1) o LSN anterior para aquela transação, (2) o ID da transação e (3) o tipo de registro de *log*. O LSN anterior é importante porque ele relaciona os registros de *log* (na sequência contrária) para cada transação. Para uma ação de atualização (*write*), campos adicionais no registro de *log* incluem: (4) o ID da página para a página que inclui o item, (5) o comprimento do item atualizado, (6) seu deslocamento a partir do início da página, (7) a imagem anterior do item e (8) sua imagem posterior.

Além do *log*, duas tabelas são necessárias para uma recuperação eficiente: a **Tabela de Transações** e a **Tabela de Páginas Lixo**, que são mantidas pelo gerenciador de transações. Quando ocorrer uma queda, essas tabelas serão reconstruídas na fase de análise da recuperação. A Tabela de Transações contém uma entrada para *cada transação ativa*, com informações como o ID da transação, o status da transação e o LSN do registro de *log* mais recente para a transação. A Tabela de Páginas Lixo contém uma entrada para cada página lixo no *buffer*, que inclui o ID da página e o LSN correspondente à atualização mais antiga para aquela página.

O *checkpoint* em ARIES consiste do seguinte: (1) escrever um registro de *begin_checkpoint* no *log*, (2) escrever um registro de *end_checkpoint* no *log* e (3) escrever o LSN do registro de *begin_checkpoint* em um arquivo especial. Esse arquivo especial é acessado durante a recuperação para localizar a última informação de *checkpoint*. Com o registro de *end_checkpoint*, o conteúdo da Tabela de Transações e da Tabela de Páginas Lixo será adicionado ao fim do *log*. A fim de reduzir o custo é usado o **fuzzy checkpoint** (ponto de controle indistinto), para que o SGBD possa continuar a executar transações durante a confecção do *checkpoint* (Seção 19.1.4). Além disso, os conteúdos do *cache* do SGBD não foram expelidos para o disco durante o *checkpoint*, uma vez que a Tabela de Transações e a Tabela de Páginas Lixo — que serão adicionadas ao *log* no disco — contém as informações necessárias para a recuperação. Observe que, se ocorrer uma queda durante a confecção do *checkpoint*, o arquivo especial irá se referir ao *checkpoint* anterior, que será usado na recuperação.

Após uma queda, o gerenciador de recuperação ARIES obterá o controle. As informações do último *checkpoint* serão acessadas primeiro, por meio do arquivo especial. A **fase de análise** inicia o registro *begin_checkpoint* e avança para o fim do *log*. Quando o registro *end_checkpoint* for encontrado, a Tabela de Transações e a Tabela de Páginas Lixo serão acessadas (lembre-se de que essas tabelas foram escritas no *log* durante o *checkpoint*). Durante a análise, o registro de *log* que está sendo estudado poderá causar modificações nessas duas tabelas. Por exemplo, se um registro de término do *log* for encontrado para uma transação T da Tabela de Transações, então a entrada para T será removida dessa tabela. Se algum outro tipo de registro for encontrado no *log* para uma transação T', então uma entrada para T' será inserida na Tabela de Transações, se já não estiver presente, e o último campo LSN será modificado. Se o registro de *log* corresponder a uma mudança na página P, então seria feita uma entrada para a página P (se ainda não estiver presente na tabela), e o campo LSN associado seria modificado. Quando a fase de análise estiver completa, as informações necessárias para REDO e UNDO terão sido compiladas nas tabelas.

6 Os *buffers* podem ser perdidos durante uma queda, uma vez que estão na memória principal. Tabelas adicionais armazenadas no *log* durante o *checkpoint* (Tabela de Páginas Lixo, Tabela de Transações) permitem ao ARIES identificar essas informações.

450 Capítulo 19 Técnicas de Recuperação de Banco de Dados

Segue a fase REDO. Para reduzir o trabalho desnecessário, o ARIES começa refazendo a partir do ponto do *log* onde ele sabe (com toda a certeza) que as mudanças anteriores para as páginas lixo já foram aplicadas ao banco de dados em disco. Ele poderá determinar isso encontrando o menor LSN, M , de todas as páginas lixo da Tabela de Páginas Lixo, o que indicará a posição de *log* onde o ARIES deve começar a fase REDO. Quaisquer mudanças relativas a $LSN < M$, para transações de REDO, já foram propagadas para o disco, ou já foram sobrescritas no *buffer*; de outro modo, essas páginas lixo, com esse tipo de LSN, estariam no *buffer* (e na Tabela de Páginas Lixo). Assim, o REDO inicia no registro de *log* com $LSN = M$, examinando a partir daí até o fim do *log*. Para cada mudança registrada no *log*, o algoritmo REDO verificará se a mudança deverá ou não ser reaplicada. Por exemplo, se uma mudança registrada no *log* pertencer à página P que não está na Tabela de Páginas Lixo, então essa mudança já estará no disco e não precisará ser reaplicada. Ou, se uma mudança registrada no *log* (digamos, com $LSN = N$) pertencer à página P , e a Tabela de Páginas Lixo contiver uma entrada para P com LSN maior que N , então a mudança já estará presente. Se nenhuma dessas duas condições existir, a página P será lida do disco e o LSN armazenado nessa página, $LSN(P)$, será comparado com N . Se $N < LSN(P)$, então a mudança foi aplicada e a página não necessitará ser regravada no disco.

Uma vez terminada a fase de REDO, o banco de dados estará exatamente no estado em que ele estava quando a queda ocorreu. O conjunto das transações ativas — chamado *undo_set* — foi identificado na Tabela de Transações durante a fase de análise. Assim, a fase UNDO continua examinando o *log* de trás para a frente e desfazendo as ações apropriadas. Um registro de *log* de compensação será gravado para cada uma das ações desfeitas. O UNDO lerá os registros anteriores do *log* até que cada ação do conjunto de transações no *undo_set* tenha sido desfeita. Ao final, o processo de recuperação estará encerrado e o processamento normal poderá ser reiniciado.

Considere o exemplo de recuperação mostrado na Figura 19.6. Há três transações: T_1 , T_2 e T_3 . T_1 atualiza a página C , T_2 atualiza as páginas B e C e T_3 atualiza a página A . A Figura 19.6a mostra os conteúdos parciais do *log* e a Figura 19.6b mostra os conteúdos da Tabela de Transações e da Tabela de Páginas Lixo. Agora suponha que ocorra uma queda nesse ponto.

(a)

(b)

(c)

LSN	ÚLTIMO LSN	I DETRAN	TIPO	ID_PAGINA	OUTRA
INFORMAÇÃO					
1	0	T_1	update	C	
2	0	T_2	update	B	
3	1	T_1	commit		
4	início checkpoint				
5	fim checkpoint				
6	0	T_3	update	A	
7	2	T_2	update	C	
8	7	T_2	commit	...	
TABELA DE TRANSAÇÕES					
ID	TRANSAÇÃO	ÚLTIMO LSN	STATUS	ID_PAGINA	LSN
	T_1	3	commit	C	1
	T_2	2	em progresso	B	2
TABELA DE TRANSAÇÕES					
ID	TRANSAÇÃO	ÚLTIMO LSN	STATUS	ID_PAGINA	LSN
	T_1	3	commit	C	1
	T_2	8	commit	B	2
	T_3	6	em progresso	A	6

FIGURA 19.6 Um exemplo de recuperação em ARIES, (a) O *log* no ponto de queda do sistema, (b) As Tabelas de Transações e de Páginas Lixo no instante do *checkpoint*, (c) As Tabelas de Transações e de Páginas Lixo depois da fase de análise.

Uma vez que um *checkpoint* tenha ocorrido, o endereço do registro associado ao *begin_checkpoint* será recuperado — e sua localização é 4. A fase de análise inicia-se na localização 4 até alcançar o fim. O registro *end_checkpoint* deverá conter a Tabela de Transações e a Tabela de Páginas Lixo, na Figura 19.6b, e a fase de análise reconstruirá, mais adiante, essas tabelas. Quando a fase de análise encontrar o registro 6 de *log*, uma nova entrada para a transação T_3 será feita na Tabela de Transações, e uma nova entrada para a página A será feita na Tabela de Páginas Lixo. Depois que o registro 8 do *log* for analisado, o status da transação T_2 será mudado para efetivado na Tabela de Transações. A Figura 19.6c mostra as duas tabelas depois da fase de análise. Para a fase REDO, o menor LSN na Tabela de Páginas Lixo é 1. Portanto, o REDO iniciará no registro 1 do *log* e continuará com o REDO de atualizações. Os LSNs 1, 2, 6, 7) correspondentes às atualizações para as páginas C , B , A e C , respectivamente,

19.6 Recuperação em Sistemas Banco de Dados Múltiplos 451

não são menores que os LSNs daquelas páginas (como mostrado na Tabela de Página Lixo). Assim, aquelas páginas de dados serão lidas novamente e as atualizações do *log*, replicadas (assumindo que os LSNs reais armazenados naquelas páginas de dado são menores que a entrada de *log* correspondente). Nesse ponto, a fase REDO será encerrada e terá início a fase UNDO. Da Tabela de Transações (Figura 19.6c), o UNDO será aplicado apenas na transação T_3 ativa. A fase UNDO se inicia na entrada 6 do log (a última atualização de T_3) e segue para trás no log. A sequência de atualizações para a transação T_1 (apenas o registro 6 do *log*, neste exemplo) será seguida e desfeita.

19.6 RECUPERAÇÃO EM SISTEMAS DE BANCOS DE DADOS MÚLTIPLOS

Até aqui assumimos implicitamente que uma transação acessa um único banco de dados. Em alguns casos, uma única transação, chamada **transação multibanco de dados**, pode exigir acesso a diversos bancos de dados. Esses bancos de dados podem, ainda, estar armazenados em diferentes tipos de SGBDs; por exemplo, alguns SGBDs podem ser relacionais, enquanto outros são orientados a objetos, hierárquicos ou SGBDs em rede. Nesse caso, cada SGBD envolvido na transação multibanco de dados pode ter sua própria técnica de recuperação, e o gerenciador de transação separa este dos outros SGBDs. Essa situação é similar ao sistema de gerenciamento de banco de dados distribuídos (Capítulo 25), no qual partes do banco de dados residem em diferentes locais, conectados por uma rede de comunicação.

Para manter a atomicidade de uma transação em um multibanco de dados, é necessário ter um mecanismo de recuperação em dois níveis. Um **gerenciador de recuperação global, ou coordenador**, será necessário para manter as informações exigidas para a recuperação, além dos gerenciadores de recuperação locais e das informações mantidas por eles (*log*, tabelas). O coordenador geralmente segue um protocolo chamado **protocolo de efetivação em duas** fases, cujas duas fases podem ser definidas assim:

- **Fase 1:** Quando todos os bancos de dados participantes sinalizarem ao coordenador que sua parte na transação multibanco foi concluída, o coordenador envia uma mensagem 'preparar para efetivar' a fim de preparar cada participante para a efetivação da transação. Cada banco de dados participante que receber essa mensagem forçará a gravação de todos os registros de *log*, bem como as informações necessárias para a recuperação local para o disco e, então, enviará um sinal de 'pronto para efetivar' ou 'OK' ao coordenador. Se a gravação forçada no disco falhar, ou se a transação local não puder ser efetivada por alguma razão, o banco de dados participante envia um sinal 'não pode efetivar' ou 'não OK' ao coordenador. Se o coordenador não receber uma resposta de um dos bancos de dados dentro de um certo intervalo de tempo, ele assumirá um 'não OK' como resposta.

- **Fase 2:** Se *todos* os bancos de dados participantes responderem 'OK' e o voto do coordenador também for um 'OK', a transação será bem-sucedida, e o coordenador envia um sinal 'efetivar' para a transação dos bancos de dados participantes. Como todos os efeitos locais da transação e as informações necessárias para recuperação local foram registrados nos *logs* dos bancos de dados participantes, a recuperação de falhas é possível. Cada banco de dados participante completa a efetivação da transação escrevendo uma entrada [efetivar] para a transação no *log* e, se necessário, atualizando permanentemente o banco de dados. De outro modo, se um ou mais dos bancos de dados participantes, ou se o coordenador optar por um 'não OK' como resposta, a transação falhará e o coordenador enviará uma mensagem para 'reverter' ou DESFAZER o efeito local da transação para cada banco de dados participante. Esse procedimento é feito desfazendo as operações de transação por meio do *log*.

O efeito em rede do protocolo de efetivação em duas fases consiste em que todos os bancos de dados participantes efetivem a transação, ou nenhum deles o faça. No caso de algum dos participantes — ou o coordenador — falhar, é sempre possível recuperar de um estado onde a transação possa ser efetivada ou revertida. Uma falha durante, ou anterior, a Fase 1 geralmente requer que as transações sejam revertidas, enquanto uma falha durante a Fase 2 significa que uma transação bem-sucedida pode ser recuperada e efetivada.

19.7 BACKUP DE BANCO DE DADOS E RECUPERAÇÃO EM FALHAS CATASTRÓFICAS

Até aqui, todas as técnicas vistas se aplicam a falhas não catastróficas. Foi feita uma suposição fundamental: que o *log* do sistema seja mantido no disco e não seja perdido como resultado da falha. Analogamente, o catálogo *shadow* deve estar armazenado em disco para permitir a recuperação se a paginação *shadow* estiver sendo usada. As técnicas de recuperação que temos

discutido usam as entradas na *log* do sistema ou o catálogo *shadow* para se recuperar de falhas, trazendo o banco de dados de volta a um estado consistente.

O gerenciador de recuperação de um SGBD deverá também estar equipado para manipular falhas catastróficas, tais como quebras de disco. A principal técnica usada nesses casos é a de backup do banco de dados. Todo o banco de dados e seu *log* são periodicamente copiados para um meio de armazenamento barato, como as fitas magnéticas. No caso de uma falha catastrófica do sistema, a última cópia de backup poderá ser recarregada, da fita para o disco, e o sistema poderá ser reiniciado.

Para evitar a perda dos efeitos das transações executadas desde o último backup, é comum salvar o *log* do sistema em intervalos menores que o usado no backup total do banco de dados, copiando-o periodicamente para a fita magnética. O *log* do sistema é, em geral, substancialmente menor que o próprio banco de dados e pode, portanto, ser copiado mais freqüentemente. Com isso, os usuários não perdem todas as transações que executaram desde o último backup do banco de dados. Todas as transações efetivadas registradas na porção do *log* de sistema que tenham sido copiadas para uma fita poderão ter seus efeitos refeitos no banco de dados. Um novo *log* será iniciado depois de cada backup do banco de dados. Portanto, para se recuperar de uma perda de disco, o banco de dados deverá ser, primeiro, recriado a partir do último backup em fita. A seguir, todas as transações efetivadas, cujas operações tenham sido registradas nas cópias backup do *log* do sistema, serão reconstruídas.

19.8 RESUMO

Neste capítulo discutimos as técnicas para recuperação de falhas de transação. O objetivo principal da recuperação é garantir a atomicidade de uma transação. Se uma transação falhar antes de completar sua execução, o mecanismo de recuperação deve garantir que ela não tenha efeitos duradouros no banco de dados. Fizemos, primeiro, um esboço informal do processo de recuperação e, depois, discutimos os conceitos para recuperação, que incluíram uma discussão de memória *cache*, atualização local versus *shadowing*, imagem anterior e posterior de um item de dado, operações de recuperação UNDO versus REDO, políticas roubada/não-roubada (*steal/no-steal*) e forçada/não-forçada (*force/no-force*), confecção de checkpoint do sistema e protocolo para registro adiantado no *log*.

Em seguida discutimos duas abordagens diferentes para recuperação: atualização adiada e atualização imediata. As técnicas de atualização adiada adiam qualquer atualização real do banco de dados em disco até que a transação alcance seu ponto de efetivação. A transação força a gravação do *log* no disco antes de registrar as atualizações no banco de dados. Essa abordagem, quando usada com certos métodos de controle de concorrência, é planejada para nunca exigir que uma transação seja revertida, e a recuperação consiste, simplesmente, em refazer, a partir do *log*, as operações das transações efetivadas depois do último *checkpoint*. A desvantagem é que pode ser necessário muito espaço para o *buffer*, uma vez que as atualizações são mantidas nos *buffers* e não são aplicadas no disco até que a transação seja efetivada. Atualização adiada pode levar a um algoritmo de recuperação conhecido como NO-UNDO/REDO. Técnicas de atualização imediata podem aplicar mudanças no banco de dados em disco antes que a transação alcance uma conclusão bem-sucedida. Qualquer mudança aplicada no banco de dados deve, primeiro, ser registrada no *log*, e sua gravação forçada no disco; assim, se necessário, essas operações poderão ser desfeitas. Também demos uma visão de um algoritmo de recuperação com atualização imediata, conhecido por UNDO/REDO. Um outro algoritmo, conhecido por UNDO/NO-REDO, também pode ser desenvolvido para atualização imediata, se todas as ações da transação forem registradas no banco de dados antes que ela seja efetivada.

Discutimos a técnica de paginação *shadow* para recuperação, que mantém o controle das páginas do banco de dados usando um catálogo *shadow*. Essa técnica, que é classificada como NO-UNDO/NO-REDO, não exige um *log* nos sistemas monousuários, mas necessita dele em sistemas multiusuários. Também apresentamos ARIES, um esquema específico de recuperação usado em alguns dos produtos de banco de dados relacionais da IBM. Depois, discutimos o protocolo de efetivação em duas fases, que é usado para recuperação de falhas envolvendo transações em bancos de dados múltiplos. Finalmente, vimos recuperação em falhas catastróficas, que normalmente é feita copiando-se o banco de dados e o *log* para uma fita. O *log* pode ser copiado com maior freqüência que o banco de dados, e o backup do *log* pode ser usado para refazer as operações a partir do último backup do banco de dados.

Questões para Revisão

- 19.1. Discuta os diferentes tipos de falhas de transação. O que se entende por falha catastrófica?
- 19.2. Discuta as ações tomadas pelas operações *ler_item* e *escreverjtem* em um banco de dados.
- 19.3. (Revisão do Capítulo 17) Para que é usado o *log* de sistema? Quais são as entradas típicas de um *log* de sistema? O que são *checkpoints* e por que eles são importantes? O que são os pontos de efetivação e por que eles são importantes?
- 19.4. Como as técnicas de *bufferização* e *cache* são usadas pelo subsistema de recuperação?

19.8 Resumo 453

- 19.5. O que é imagem anterior (BFIM) e imagem posterior (AFIM) de um item de dado? Qual é a diferença entre atualização local e *shadowing* com respeito às manipulações das BFIM e AFIM?
- 19.6. O que são as entradas de *log* do tipo UNDO e REDO?
- 19.7. Descreva o protocolo de registro adiantado em *log*.
- 19.8. Identifique três listas de transações que normalmente são mantidas pelo subsistema de recuperação.
- 19.9. O que se entende por reversão (*rollback*) de transação? O que se entende por reversão (*rollback*) em cascata? Por que os métodos práticos de recuperação usam protocolos que não permitem reversão em cascata? Quais dentre as técnicas de recuperação não exigem nenhuma reversão?
- 19.10. Discuta as operações UNDO e REDO e as técnicas de recuperação que cada uma usa.
- 19.11. Discuta a técnica de recuperação com atualização adiada. Quais são as vantagens e as desvantagens dessa técnica? Por que ela é chamada método NO-UNDO/REDO?
- 19.12. Como pode ser tratada a recuperação de operações de uma transação que não afeta o banco de dados, como uma transação para a impressão de relatório?
- 19.13. Discuta a técnica de recuperação por atualização imediata em ambos os ambientes, monousuário e multiusuário. Quais são as vantagens e as desvantagens da atualização imediata?
- 19.14. Qual é a diferença entre os algoritmos para recuperação UNDO/REDO e UNDO/NO-REDO com atualização imediata? Desenvolva um esboço para um algoritmo UNDO/NO-REDO.
- 19.15. Descreva a técnica de recuperação de paginação *shadow*. Sob quais circunstâncias ela não requer um *log*?
- 19.16. Descreva as três fases do método de recuperação ARIES.
- 19.17. O que são os números de seqüência de *log* (LSNs) em ARIES? Como eles são usados? Quais informações contêm a Tabela de Páginas Lixo e a Tabela de Transações? Descreva como o *fuzzy checkpointing* é usado em ARIES.
- 19.18. O que significam os termos roubada/não-roubada (*steal/no-steal*) e forçada/não-forçada (*force/no-force*) com respeito ao gerenciador de *buffer* no processamento de transações?
- 19.19. Descreva o protocolo de efetivação em duas fases para transações em bancos de dados múltiplos.
- 19.20. Discuta como é o tratamento para recuperação de falhas catastróficas.

Exercícios

- 19.21. Suponha que haja uma queda no sistema antes que a entrada [ler_item, T₃, A] seja escrita no *log* da Figura 19.1b. Fará alguma diferença no processo de recuperação?
- 19.22. Suponha que o sistema caia antes que a entrada [escrever_item, T₂, D, 25, 26] seja gravada no *log* da Figura 19.1b. Fará alguma diferença no processo de recuperação?
- 19.23. A Figura 19.7 mostra o *log* correspondente a determinado plano, para quatro transações T₁, T₂, T₃ e T₄, no ponto da queda do sistema. Suponha que usemos o *protocolo de atualização imediata com checkpoint*. Descreva o processo para recuperação da queda do sistema. Especifique quais transações serão revertidas, quais operações do *log* serão refeitas e quais (se houver) serão desfeitas, e se poderá ocorrer alguma reversão em cascata.
- 19.24. Suponha que usemos o protocolo de atualização adiada do exemplo da Figura 19.7. Mostre como o *log* seria, com a atualização adiada, após a remoção das entradas desnecessárias do *log*; depois, descreva o processo de recuperação usando seu *log* modificado. Suponha que apenas as operações REDO sejam aplicadas e especifique quais operações no *log* serão refeitas e quais serão ignoradas.
- 19.25. Como o registro de *checkpoint* no ARIES difere do *checkpoint* descrito na Seção 19.1.4?
- 19.26. Como o uso dos números de seqüência de *log* do ARIES reduzem a quantidade de trabalho REDO na recuperação? Ilustre com um exemplo usando as informações mostradas na Figura 19.6. Você pode fazer suas próprias suposições, como quando uma página deverá ser gravada em disco.
- 19.27. Quais implicações teria uma política de gerenciamento não-roubada/forçada (*no-steal/force*) de *buffer* para a confecção do *checkpoint* e da recuperação?
- Escolha a resposta correta para cada uma das seguintes questões de múltipla escolha:
- 19.28. O *log* incremental com atualização adiada implica que o sistema de recuperação deva, necessariamente
- armazenar o valor antigo do item atualizado no *log*.
 - armazenar o valor novo do item atualizado no *log*.
 - armazenar ambos os valores do item, o antigo e o novo, no *log*.
 - armazenar apenas os registros de Início da Transação e da Efetivação da Transação no *log*.

454

Capítulo 19 Técnicas de Recuperação de Banco de Dados

[inicia_transacao, 7,]

[ler_item, T_i , A][ler_item, T_i , D][escrever_item, T_i , D, 20, 25]

[confirma, 7",]

[checkpoint]

[inicia_transacao, 7₂][ler_item, T_2 , B][escrever_item, T_2 , B, 12, 18]

[inicia_transacao, 7J]

[ler_item, 7₄, D][escrever_item, 7₄, D, 25, 15][inicia_transacao, 7₃][escrever_item, T_w , C, 30, 40][ler_item, 7₄, A][escrever_item, 7₄, A, 30, 20]

[confirma, 7J]

[ler_item, 7₂, D][escrever_item, 7₂, D, 15, 25] <- queda do sistemaFIGURA 19.7 Um plano de execução exemplo e seu *log* correspondente.19.29. O protocolo registro adiantado em *log* (*write-ahead logging* — WAL) significa simplesmente que

- gravação de um item de dado seria feita antes de qualquer operação de *log*.
- o registro em *log* de uma operação seria gravado antes do dado propriamente dito.
- todos os registros de *log* seriam gravados antes que uma nova transação inicie sua execução.
- o *log* não precisaria nunca ser gravado no disco.

19.30. No caso da transação falhar sob um esquema de *log* incremental de atualização adiada, qual das seguintes opções seria necessária:

- uma operação desfazer.
- uma operação refazer.
- uma operação desfazer e refazer.
- nenhuma acima.

19.31. Para um *log* incremental com atualizações imediatas, um registro de ítem uma transação poderia conter:

- um nome de transação, nome do item de dado, valor antigo do item, valor novo do item.
- um nome de transação, nome do item de dado, valor antigo do item.
- um nome de transação, nome do item de dado, valor novo do item.
- um nome de transação e um nome do item de dado.

19.32. Para comportamento correto durante a recuperação, as operações desfazer e refazer devem ser

- comutativas.
- associativas.
- potencialmente idênticas.
- distributivas.

19.33. Quando ocorre uma falha, o *log* é consultado e cada operação será desfeita ou refeita. Isso porque

- a pesquisa no *log* completo consome muito tempo.
- alguns dos 'refazer' são desnecessários.
- (a) e (b) estão corretos.
- nenhuma acima.

19.34. Quando se usa algum esquema de recuperação baseado em *log*, pode-se melhorar o desempenho da execução e implementar um mecanismo de recuperação

- escrevendo os registros de *log* no disco quando cada transação for efetivada.
- escrevendo os registros apropriados de *log* no disco durante a execução da transação.
- esperando para escrever os registros de *log* quando diversas transações forem efetivadas e escrevê-los em lote.
- nunca escrever os registros de *log* no disco.

19.35. Há possibilidade de reversão (*rollback*) em cascata quando

- uma transação grava itens que tenham sido gravados apenas por uma transação efetivada.
- uma transação grava um item que já foi previamente gravado por uma transação não-efetivada.

19.8 Resumo | 455

- c. uma transação lê um item que foi previamente gravado por uma transação não-efetivada.
- d. ambos (b) e (c).

19.36. Para enfrentar falhas de mídia (disco), é necessário

- a. o SGBD executar transações apenas em um ambiente monousuário.
- b. manter uma cópia redundante do banco de dados.
- c. nunca abortar uma transação.
- d. todas acima.

19.37. Se uma abordagem *shadowing* for usada para gravar um item de dado de volta no disco, então

- a. o item será gravado no disco apenas depois que a transação for efetivada.
- b. o item será gravado em uma localização diferente do disco.
- c. o item será gravado no disco antes da transação ser efetivada.
- d. o item será gravado na mesma localização do disco da qual ele foi lido.

Bibliografia Seleccionada

Os livros de Bernstein et al. (1987) e Papadimitriou (1986) são dedicados à teoria e aos princípios de controle de concorrência e recuperação. O livro de Gray e Reuter (1993) é um compêndio sobre controle de concorrência, recuperação e outras opções sobre processamento de transações.

Verhofstad (1978) apresenta um tutorial e uma pesquisa sobre as técnicas de recuperação em sistemas de banco de dados.

Algoritmos de categorização UNDO/REDO, baseados em suas características, são discutidos em Haerder e Reuter (1983) e em Bernstein et al. (1983). Gray (1978) discute recuperação e outros aspectos de implementação entre sistemas operacionais e implementação de banco de dados. A técnica de paginação *shadowing* é discutida em Lorie (1977), Verhofstad (1978) e Reuter (1980). Gray et al. (1981) discutem o mecanismo de recuperação no SYSTEM R. Lockeman e Knutsen (1968), Davies (1972) e Bjork (1973) são alguns dos primeiros autores que discutem recuperação. Chandy et al. (1975) discutem reversão de (*rollback*) transação. Lilien e Bhargava (1985) discutem o conceito de bloco de integridade e seu uso para melhorar a eficiência da recuperação.

Recuperação usando registro adiantado em *log (write-ahead logging)* é analisado em Jhingran e Khedkar (1992) e é usado no sistema ARIES (Mohan et al., 1992a). Um trabalho recente em recuperação engloba transações de compensação (Korth et al., 1990) e recuperação de banco de dados da memória principal (Kumar, 1991). Os algoritmos de recuperação ARIES (Mohan et al., 1992) têm tido muito sucesso na prática. Franklin et al. (1992) discutem recuperação no sistema EXODUS. Dois livros recentes, de Kumar e Hsu (1998) e Kumar e Son (1998), analisam recuperação em detalhes e descrevem métodos de recuperação usados em diversos produtos existentes de banco de dados relacional.

•III

6

Bancos de Dados de Objetos e Objeto-
Relacionais

20

Conceitos para Bancos de Dados de Objetos

Neste capítulo e no próximo serão abordados modelos e sistemas de bancos de dados orientados a objetos.¹ Modelos de dados e sistemas tradicionais, como relacionai, rede e hierárquico, foram muito bem-sucedidos no desenvolvimento da tecnologia de banco de dados necessária para a maioria das aplicações convencionais de bancos de dados comerciais. Entretanto, eles têm algumas limitações quando aplicações de bancos de dados mais complexas devem ser projetadas e implementadas — como, por exemplo, bancos de dados para projetos de engenharia e manufatura (CAD/CAM e CIM), experimentos científicos, telecomunicações, sistemas de informações geográficas e multimídia. Essas novas aplicações têm requisitos e características que as diferenciam das tradicionais aplicações comerciais, como estruturas complexas para objetos, transações de longa duração, novos tipos de dados para armazenamento de imagens ou textos longos, e a necessidade de se definir operações não convencionais específicas da aplicação. Os bancos de dados orientados a objetos foram propostos para atender às necessidades dessas aplicações mais complexas. A abordagem orientada a objetos oferece a flexibilidade para lidar com alguns desses requisitos sem estar limitada pelos tipos de dados e linguagens de consulta disponíveis em sistemas de bancos de dados tradicionais. Uma característica-chave dos bancos de dados orientados a objetos é o poder dado ao projetista para especificar tanto a *estrutura* de objetos complexos quanto as *operações* que podem ser aplicadas a esses objetos.

Uma outra razão para a criação de bancos de dados orientados a objetos é o uso crescente de linguagens de programação orientadas a objetos no desenvolvimento de aplicações de software. Os bancos de dados vêm se tornando componentes fundamentais em muitos sistemas de software, e os bancos de dados tradicionais são difíceis de ser utilizados em aplicações de software que são desenvolvidas em uma linguagem de programação orientada a objetos, como C++, SMALLTALK ou JAVA. Bancos de dados orientados a objetos são projetados de modo que possam ser diretamente — ou *de maneira unificada* — integrados com o software que está sendo desenvolvido e que utiliza linguagens de programação orientadas a objetos.

A necessidade de características adicionais na modelagem de dados também tem sido reconhecida por fornecedores de SOBDs relacionais, e as recentes versões de sistemas relacionais estão incorporando muitas das características que foram propostas para bancos de dados orientados a objetos, o que resultou em sistemas conhecidos como SGBDs *objeto-relacionais* ou *relacionais estendidos* (Capítulo 22). A última versão do padrão SQL para bancos de dados relacionais inclui algumas dessas características.

¹ Estes bancos de dados são freqüentemente referenciados como bancos de dados de objetos e os sistemas como sistemas gerenciadores de bancos de dados de objetos (SGBDOs). Porém, por este capítulo discutir muitos conceitos gerais de orientação a objetos, será utilizado o termo *orientado a objetos* em vez de somente *objeto*.

² Computer-Aided Design (Projeto Auxiliado por Computador)/Computer-Aided Manufacturing (Fabricação Auxiliada por Computador) e Computer-Integrated Manufacturing (Fabricação Integrada por Computador).

³ Bancos de dados multimídia devem armazenar vários tipos de objetos multimídia, como vídeo, áudio, imagens, gráficos e documentos (Capítulo 24).

Capítulo 20 Conceitos para Bancos de Dados de Objetos

Apesar de terem sido criados vários protótipos experimentais e versões comerciais de sistemas de bancos de dados orientados a objetos, eles não foram utilizados em larga escala por causa da popularidade de sistemas de banco de dados relacionais. Dentre os protótipos experimentais podem-se citar o sistema ORION, desenvolvido na MCC ; OPENOOD na Texas Instruments; o sistema IRIS, nos laboratórios Hewlett-Packard; o sistema ODE, no AT&T Bell Labs ; e o projeto ENCORE ObServer, na Universidade de Brown. Os sistemas disponíveis comercialmente incluem o GEMSTONE/OPAL, < GemStone Systems; ONTOS, da Ontos; Objectivity, da Objectivity Inc.; Versant, da Versant Object Technology; Objec Store, da Object Design; ARDENT, da ARDENT Software ; e POET, da POET Software. Eles representam apenas uma lista pareia de protótipos experimentais e sistemas comerciais de bancos de dados orientados a objetos que foram criados.

A medida que os SGBDs comerciais orientados a objetos tornaram-se disponíveis, sentiu-se necessidade de um modelo de linguagem padrão. Uma vez que o procedimento formal para aprovação de padrões geralmente demanda alguns anos e um consórcio de fornecedores e usuários de SGBDs orientados a objetos, denominado ODMG, propôs um padrão que é conhecido como padrão ODMG-93, que, desde então, tem sido revisado. Algumas características do padrão ODMG serão descritas no Capítulo 21.

Os bancos de dados orientados a objetos adotaram muitos conceitos que foram desenvolvidos inicialmente para as li-

nguagens de programação orientadas a objetos. Na Seção 20.1 serão examinadas as origens da abordagem orientada a objetos discutidas suas aplicações em sistemas de bancos de dados. A seguir, nas seções 20.2 a 20.6, serão descritos os principais conceitos utilizados na maioria dos sistemas de bancos de dados orientados a objetos. A Seção 20.2 discute *identidade de objetos*, *estrutura de objetos* e *construtores de tipos*. A Seção 20.3 apresenta os conceitos de *encapsulamento de operações* e a definição de *método* como parte da declaração de classes, além de discutir os mecanismos para armazenamento de objetos em bancos de dados tornando-os *persistentes*. A Seção 20.4 descreve *tipo*, *hierarquias de classes* e *herança* em bancos de dados orientados a objetos. A Seção 20.5 apresenta uma visão geral das questões que surgem quando *objetos complexos* precisam ser representados e armazenados. A Seção 20.6 discute conceitos adicionais, incluindo *polimorfismo*, *sobrecarga de operador*, *acoplamento dinâmico* (*dynamic binding*), *herança múltipla e seletiva*, *versionamento* e *configurações* de objetos.

Este capítulo apresenta os conceitos gerais sobre bancos de dados orientados a objetos, enquanto o Capítulo 22 apresentará o padrão ODMG. O leitor poderá abreviar as seções 20.5 e 20.6 se desejar uma introdução menos detalhada deste tópico.

20.1 VISÃO GERAL DOS CONCEITOS DE ORIENTAÇÃO A OBJETOS

Esta seção fornece uma rápida visão geral sobre a história e os principais conceitos de bancos de dados orientados a objetos ou BDOO. Os conceitos de BDOO serão explicados mais detalhadamente nas seções 20.2 a 20.6. O termo *orientado a objetos* — abreviado como OO ou O-O — tem sua origem nas linguagens de programação OO, ou LPOOs. Atualmente, os conceitos CK são aplicados nas áreas de bancos de dados, engenharia de software, bases de conhecimento, inteligência artificial e sistemas computacionais em geral. As LPOOs têm suas raízes na linguagem SIMULA, que foi proposta no final dos anos 60. Na SIMULA o conceito de uma classe agrupa a estrutura de dados interna de um objeto em uma declaração de classe. Subseqüentemente os pesquisadores propuseram o conceito de *tipo abstrato de dados*, que oculta as estruturas internas de dados e especifica toda as possíveis operações externas aplicáveis a um objeto, conduzindo ao conceito de *encapsulamento*. A linguagem de programação SMALLTALK, desenvolvida pela Xerox PARC nos anos 70, foi uma das primeiras a incorporar explicitamente conceitos de OO adicionais, como troca de mensagens e herança. Conhecida como linguagem de programação OO *pura*, ela foi explicitamente projetada para ser orientada a objetos, contrapondo-se às linguagens de programação OO *híbridas*, que incorporam conceitos OO em uma linguagem já existente. Um exemplo dessa última é C++ , a qual incorpora conceitos OO à popular linguagem de programação C.

Um objeto possui, tipicamente, dois componentes: estado (valor) e comportamento (operações). Assim, é similar a uma *variável de programa* em uma linguagem de programação, exceto que geralmente terá uma *estrutura de dados complexa* bem como *operações específicas* definidas pelo programador. Objetos em uma LPOO existem somente durante a execução de programa e são chamados *objetos transientes*. Um banco de dados OO pode estender a existência de objetos de modo que sejam

4 Microelectronics and Computer Technology Corporation, Austin, Texas.

5 Atualmente Lucent Technologies.

6 Anteriormente O2, da O2 Technology.

7 Object Database Management Group. (Grupo de Gerenciamento de Bancos de Dados de Objetos.)

8 Conceitos similares também foram desenvolvidos nas áreas de modelagem semântica de dados e representação do conhecimento.

9 Centro de Pesquisas de Páio Alto — Páio Alto, Califórnia.

10 Objetos possuem muitas outras características, como será discutido no restante deste capítulo.

20.1 Visão Geral dos Conceitos de Orientação a Objetos

461

armazenados de forma permanente; portanto, os objetos continuam após o término do programa, podendo ser posteriormente recuperados e compartilhados por outros programas. Em outras palavras, bancos de dados OO armazenam *objetos persistentes* permanentemente, em memória secundária, permitindo o compartilhamento desses objetos entre vários programas e aplicações. Isso exige incorporar outras características bem conhecidas de sistemas gerenciadores de bancos de dados, como mecanismos de indexação, controle de concorrência e recuperação. Um sistema de banco de dados possui interface com uma ou mais linguagens de programação OO para fornecer as capacidades de objetos persistentes e compartilhados.

Um dos objetivos dos bancos de dados OO é manter a correspondência direta entre objetos do mundo real e do banco de dados, assim esses objetos não perdem sua integridade e identidade e podem ser facilmente identificados e acessados. Então, o banco de dados OO fornece um *identificador do objeto* (OID), único, gerado pelo sistema para cada objeto. Podemos comparar esta funcionalidade com o modelo relacionai, no qual cada relação precisa ter uma chave primária cujo valor identifica cada tupla univocamente. No modelo relacionai, se o valor da chave primária é alterado, a tupla terá uma nova identidade, embora ela possa continuar representando o mesmo objeto do mundo real. Alternativamente, um objeto do mundo real pode ter diferentes nomes para atributos-chave em relações diferentes, tornando difícil averiguar quais chaves representam quais objetos (por exemplo, o identificador do objeto pode ser representado como ID_EMP em uma relação e como SSN em outra).

Outra característica dos bancos de dados OO é que os objetos podem ter uma *estrutura de objeto de complexidade arbitrária*, de forma a conter todas as informações necessárias que descrevem o objeto. Em contrapartida, nos sistemas de bancos de dados tradicionais, a informação sobre objetos complexos é normalmente *distribuída* em várias relações ou registros, levando à perda da correspondência direta entre um objeto do mundo real e sua representação no banco de dados.

A estrutura interna de um objeto nas LPOOs incluem a especificação de variáveis de instância, as quais mantêm os valores que definem o estado interno do objeto. Assim, uma variável de instância é semelhante ao conceito de um *atributo* no modelo relacionai, exceto que as variáveis de instância podem ser encapsuladas no objeto, não sendo necessariamente visíveis para os usuários externos. As variáveis de instância também podem ser de tipos de dados arbitrariamente complexos. Os sistemas orientados a objetos permitem a definição de operações ou funções (comportamento) que podem ser aplicadas a objetos de um tipo particular. De fato, alguns modelos OO enfatizam que todas as operações que um usuário pode aplicar a um objeto devem ser predefinidas, acarretando um *encapsulamento completo* dos objetos. Essa abordagem rígida foi abrandada na maioria dos modelos de dados OO por várias razões. Primeiro, o usuário do banco de dados normalmente necessita conhecer os nomes dos atributos de modo que possa especificar as condições de seleção sobre os atributos para recuperar objetos específicos. Segundo, o encapsulamento completo acarreta que qualquer simples recuperação requeira uma operação predefinida, tornando as consultas *ad-hoc* difíceis de serem especificadas rapidamente.

Para estimular o encapsulamento, uma operação é definida em duas partes. A primeira, chamada *assinatura* ou *interface* da operação, especifica o nome da operação e os argumentos (ou parâmetros). A segunda, chamada *método* ou *corpo*, especifica a *implementação* da operação. As operações podem ser invocadas pela passagem de uma *mensagem* a um objeto, a qual inclui o nome da operação e os parâmetros. O objeto então executa o método para aquela operação. O encapsulamento permite a modificação da estrutura interna de um objeto, bem como a implementação de suas operações, sem causar distúrbios nos programas externos que invocam essas operações. Assim, o encapsulamento dá suporte a uma forma de independência de dados e operações (Capítulo 2). Outro conceito-chave em sistemas OO é referente à *herança* e às hierarquias de tipo e classe. Este conceito permite a especificação de novos tipos ou classes que herdam parte de suas estruturas e/ou operações de classes ou tipos previamente definidos. Assim, a especificação de tipos de objetos pode ser realizada sistematicamente. Isso torna mais fácil desenvolver os tipos de dados de um sistema de modo incremental e *reutilizar* definições de tipos na criação de novos tipos de objetos.

Um problema nos primeiros bancos de dados OO envolveu a representação de *relacionamentos* entre objetos. A insistência no encapsulamento completo nos primeiros modelos de dados OO levou ao argumento de que os relacionamentos não deveriam ser explicitamente representados, mas deveriam ser descritos pela definição de métodos apropriados que localizariam os objetos relacionados. Entretanto, essa abordagem não funciona adequadamente para bancos de dados complexos com muitos relacionamentos, porque é interessante identificar esses relacionamentos e torná-los visíveis para os usuários. O padrão ODMG reconheceu essa necessidade e representa explicitamente relacionamentos binários por meio de um par de *re/e-rências inversas* — isto é, indicando os OIDs dos objetos relacionados internamente nos objetos, mantendo a integridade referencial, como será visto no Capítulo 21.

Alguns sistemas OO oferecem facilidades para lidar com *múltiplas versões* do mesmo objeto — uma característica que é essencial nas aplicações de projeto e engenharia. Por exemplo, uma versão anterior de um objeto que representa um projeto Normalmente representado por OID, derivado do inglês Object Identifier — Identificador de Objeto. (N. de R.T.)

testado e verificado deve ser retida até que uma nova versão seja testada e verificada. Uma nova versão de um objeto complexo pode incluir apenas algumas novas versões de seus objetos componentes, enquanto outros componentes permanecem inalterados. Além de permitir versões, os bancos de dados OO deveriam também permitir a *evolução de esquema*, que ocorre quando as declarações de tipos são modificadas ou quando novos tipos ou relacionamentos são criados. Essas duas características não são específicas dos BDOOs e seria ideal que fossem incluídas em todos os tipos de SGBDs.

Outro conceito OO é a *sobrecarga de operador*, que se refere à propriedade de uma operação de ser aplicada a diferentes tipos de objetos; em tal situação, um *nome de operação* pode se referir a várias *implementações* diferentes dependendo do tipo de objetos aos quais é aplicada. Essa característica é também conhecida como *polimorfismo de operador*. Por exemplo, uma operação para calcular a área de um objeto geométrico pode diferir em seu método (implementação) dependendo se o objeto é um triângulo, círculo ou retângulo. Isso pode exigir o uso de *acoplamento tardio* (*late binding*) do nome da operação com o método apropriado em tempo de execução, quando o tipo do objeto no qual a operação é aplicada torna-se conhecido.

Esta seção deu uma visão geral dos principais conceitos de bancos de dados OO. Nas seções 20.2 a 20.6 esses conceitos serão discutidos com maiores detalhes.

20.2 IDENTIDADE DE OBJETO, ESTRUTURA DE OBJETO E CONSTRUTORES DE TIPOS

Será analisado, nesta seção, o conceito de identidade de objeto e serão apresentadas as operações típicas de estruturação para definir a estrutura do estado de um objeto. Essas operações de estruturação são normalmente chamadas construtores de tipos. Elas definem as operações básicas de estruturação de dados que podem ser combinadas para construir estruturas de objetos complexos.

20.2.1 Identidade de Objeto

Um sistema de banco de dados OO fornece uma identidade única para cada objeto independente armazenado no banco de dados. Essa identidade única é geralmente implementada por meio de um identificador de objeto único gerado pelo sistema, ou OID. O valor de um OID não é visível para um usuário externo, mas é utilizado internamente pelo sistema para identificar univocamente cada objeto e para criar e gerenciar referências interobjetos. Quando necessário, o OID pode ser atribuído a variáveis de programa de tipos apropriados.

A principal propriedade exigida de um OID é que ele seja imutável, isto é, o valor do OID para um objeto particular não deve ser modificado. Isso preserva a identidade do objeto do mundo real que está sendo representado. Desse modo, um sistema de banco de dados OO deve possuir algum mecanismo para gerar os OIDs e garantir a propriedade da imutabilidade. Também é desejável que cada OID seja utilizado apenas uma vez, ou seja, mesmo que um objeto seja removido do banco de dados, seu OID não deve ser atribuído a outro objeto. Essas duas propriedades determinam que o OID não deve depender de quaisquer valores de atributos do objeto, uma vez que o valor de um atributo pode ser modificado ou corrigido. Também, em geral, é considerado inapropriado basear o OID no endereço físico de armazenamento do objeto porque este endereço pode ser modificado após uma reorganização física do banco de dados. Entretanto, alguns sistemas fazem uso do endereço físico para o OID para melhorar a eficiência da recuperação do objeto. Se o endereço físico do objeto se modifica, um *ponteiro indireto* pode ser colocado no endereço anterior, fornecendo a nova localização física do objeto. É mais comum utilizar inteiros longos como OIDs e depois usar alguma forma de tabela *hash* para mapear o valor do OID para o endereço físico de armazenamento do objeto.

Alguns dos primeiros modelos de dados OO exigiam que tudo — desde um simples valor até um objeto complexo — fosse representado como um objeto, sendo que cada valor básico, como um inteiro, uma cadeia de caracteres ou um valor lógico (Booleano), possuía um OID, permitindo que dois valores básicos tivessem diferentes OIDs, o que podia ser útil em alguns casos. Por exemplo, o valor inteiro 50 podia ser utilizado algumas vezes para representar o peso em quilos e em outras vezes para representar a idade de uma pessoa. Então, dois objetos básicos com OIDs diferentes poderiam ser criados, mas ambos os objetos representariam o valor inteiro 50. Apesar de ser útil como um modelo teórico, não é muito prático porque pode levar à geração de muitos OIDs. Desse modo, a maioria dos sistemas de bancos de dados OO permite a representação tanto de objetos como de valores. Cada objeto deve possuir um OID não modificável, enquanto um valor não possui um OID. Dessa forma, um valor normalmente é armazenado em um objeto e *não pode ser referenciado* a partir de outros objetos. Em alguns sistemas, se necessário, valores complexos estruturados também podem ser criados sem possuir um OID correspondente.

11 Várias operações de evolução de esquema, como ALTER TABLE, já estão definidas no padrão relacionai.

20.2 Identidade de Objeto, Estrutura de Objeto e Construtores de Tipos

463

20.2.2. Estrutura de Objeto

Em bancos de dados OO, o estado (valor corrente) de um objeto complexo pode ser construído a partir de outros objetos (ou outros valores) pelo uso de alguns construtores de tipos. Um modo formal de representar tais objetos é visualizar cada objeto como uma tripla (i, c, v) , na qual i é o *identificador único do objeto* (o OID), c é um *construtor de tipo* (ou seja, uma indicação de como o estado do objeto é construído) e v é o *estado do objeto* (ou *valor corrente*). O modelo de dados normalmente inclui vários construtores de tipos. Os três construtores mais básicos são **atom** (atômico), **tuple** (tupla) e **set** (conjunto). Outros construtores geralmente utilizados são **list**, **bag** e **array**. O construtor **atom** é utilizado para representar todos os valores atômicos básicos, como inteiros, números reais, cadeias de caracteres, Booleanos e quaisquer outros tipos básicos que o sistema suporte diretamente.

O estado v de um objeto (i, c, v) é interpretado com base no construtor c . Se $c = \text{atom}$, o estado (valor) v é um valor atômico do domínio de valores básicos suportados pelo sistema. Se $c = \text{set}$, o estado v é um *conjunto de identificadores de objetos* $\{i_1, i_2, \dots, i_n\}$, os quais são OIDs para um conjunto de objetos normalmente do mesmo tipo. Se $c = \text{tuple}$, o estado v é uma tupla da forma $\langle a_1:i_1, a^{\wedge}i_2, \dots, a_n:i_n \rangle$, em que cada a^{\wedge} é um nome de atributo e cada i é um OID. Se $c = \text{list}$, o valor v é uma lista ordenada $[i_1, i_2, \dots, i_j]$ de OIDs para objetos do mesmo tipo. Uma lista é similar a um conjunto com exceção de que os OIDs em uma lista estão *ordenados* e, assim, podemos nos referenciar ao primeiro, segundo ou j -ésimo em uma lista. Para $c = \text{array}$, o estado de um objeto é um vetor unidimensional de identificadores de objetos. A principal diferença entre **array** e **list** é que uma lista pode conter um número arbitrário de elementos, enquanto um **array** geralmente tem tamanho máximo. A diferença entre **set** e **bag** é que todos os elementos de um **set** devem ser distintos, enquanto em um **bag** pode haver elementos duplicados.

Este modelo de objetos permite o aninhamento arbitrário dos tipos **set**, **list**, **tuple** e outros construtores. O estado de um objeto que não seja do tipo atômico (**atom**) se referirá a outros objetos por seus identificadores de objetos. Assim, o único caso em que um valor real aparece é em um *estado de um objeto do tipo atômico*.

Os construtores de tipos **set**, **list**, **array** e **bag** são chamados **tipos coleção** (ou **tipos empilhados**) para distingui-los dos tipos básicos e dos tipos tuplas. A principal característica de um tipo coleção é que o estado de um objeto será uma *coleção de objetos* que podem ser não ordenados (como um do tipo **set** ou do tipo **bag**) ou ordenados (como uma lista ou um **array**). O construtor de tipo **tupla** é freqüentemente chamado **tipo estruturado**, uma vez que corresponde à construção **struct** nas linguagens de programação C e C++.

EXEMPLO 1: Um objeto complexo

Alguns objetos do banco de dados relacionai apresentado na Figura 5.6 serão representados utilizando o modelo proposto, em que um objeto é definido como uma tripla (OID, construtor de tipo, estado) e os construtores de tipos disponíveis são **atom**, **set** e **tuple**. Serão utilizados $i_1, i_2, i^{\wedge}, \dots$ para representar identificadores de objetos únicos gerados pelo sistema. Considere os seguintes objetos:

01 = $(i_1, \text{atom}, \text{'Houston'})$ 02 = $(i_2, \text{atom}, \text{'Bellaire'})$

03 = $(i_3, \text{atom}, \text{'Sugarland'})$

04 = $(i_4, \text{atom}, 5)$

05 = $(i_5, \text{atom}, \text{'Research'})$

06 = $(i_6, \text{atom}, \text{'1988-05-22'})$

07 = $(i_7, \text{set}, \{i_1, i_2, i_3\})$

08 = $(i_8, \text{tuple}, \langle \text{DNOME}:i_5, \text{DNUMERO}:i_4, \text{GER}:i_9, \text{LOCALIZACOES}:i_7, \text{EMPREGADOS}:i_{10}, \text{PROJETOS}:!, \rangle)$

09 = $(i_9, \text{tuple}, \langle \text{GERENTE}:i_{12}, \text{GER_DATA_INICIO}:i_6 \rangle)$

10 = $(i_{10}, \text{set}, \{i_1, i_2, i_3, i_4\})$

11 = $(i_{11}, \text{set}, \{i_{15}, i_{16}, i_{17}\})$

12 = $(i_{12}, \text{tuple}, \langle \text{PNOME}:i_{18}, \text{MINICIAL}:i_{19}, \text{UNOME}:i_{20}, \text{SSN}:i_{21}, \dots, \text{SALARIO}:i_{26}, \text{SUPERVISOR}:i_{27}, \text{DEPTO}:i_8 \rangle)$

12 Isso é diferente da operação *constructor* utilizada em C++ e outras LPOOs para criar novos objetos.

13 Também chamado de *nome de variável de instância* na terminologia OO.

14 Também chamado de multiconjunto (multiset).

15 Como foi visto anteriormente, não é prático gerar um identificador único de sistema para cada valor, assim os sistemas reais permitem que OIDs e valores *estruturados* possam ser estruturados utilizando os mesmos construtores de tipos objetos, exceto a consideração de que um valor não possui um OID.

464 Capítulo 20 Conceitos para Bancos de Dados de Objetos

Os seis primeiros objetos (o_1 – o_6) listados representam valores atômicos. Haverá muitos objetos similares, um para cada valor constante atômico diferente no banco de dados. O objeto o_7 possui valor do tipo set, que representa o conjunto de 13 localizações do departamento 5; o conjunto $\{i_1, i_2, i_3\}$ refere-se aos objetos atômicos com valores {'Houston', 'Bellaire' e 'Sugarland'}. O objeto o_8 tem valor do tipo tupla, que representa o próprio departamento 5 e possui os atributos DNOME, DNUMERO, GLOCALIZAÇÕES, e assim por diante. Os primeiros dois atributos, DNOME e DNUMERO, têm como seus valores os objetos atômicos o_5 e o_6 . O atributo GER tem o objeto do tipo tupla o_9 como seu valor, o qual, por sua vez, tem dois atributos. O valor do atributo GERER corresponde ao objeto cujo OID é i_{12} , o qual representa o empregado 'John B. Smith', que gerencia o departamento, enquanto o valor de GER_DATA_INICIO é um outro objeto atômico cujo valor é uma data. O valor do atributo EMPREGADOS de o_8 é um conjunto de objetos com $OID = i_{10}$, cujo valor é um conjunto de identificadores de objetos para os empregados que trabalham no DEPARTAMENTO (objetos i_{12} , mais i_{13} e i_{14} , que não estão mostrados). Analogamente, o valor do atributo PROJETOS de o_8 é um conjunto de objetos com $OID = i_m$, cujo valor é o conjunto dos identificadores dos objetos para os projetos que são controlados pelo departamento número 5 (objetos i_{15} , i_{16} e i_{17} , que não são mostrados). O objeto cujo $OID = i_{12}$ representa o empregado 'John B. Smith', com todos os seus atributos atômicos (PNOME, MINICIAL, UNOME, SSN, ..., SALÁRIO), que estão referenciando os objetos atômicos i_{18} , i_{19} , i_{20} , i_{21} , ..., i_{26} , respectivamente (não mostrados), mais SUPERVISOR, que referencia o objeto empregado com $OID = i_{27}$ (que representa 'James E. Borg', o qual supervisiona 'John B. Smith', mas não é mostrado), e DEPTO, que referencia o objeto departamento com $OID = i_8$ (que representa o departamento número 5 em que 'John B. Smith' trabalha).

Neste modelo, um objeto pode ser representado como uma estrutura de grafo que pode ser construída recursivamente aplicando-se os construtores de tipos. O grafo representando o objeto o_i pode ser construído inicialmente pela criação de um nó partilhado próprio objeto o_i . O nó para o_i é rotulado com o OID e o construtor c . Também é criado um nó no grafo para cada valor atômico específico. Se o objeto o_i tem um valor atômico, é desenhado um arco direcionado do nó representando o_i até o nó que representa seu valor básico. Se o valor do objeto for construído, são desenhados os arcos a partir do nó do objeto até um nó que representa o valor construído. A Figura 20.1 mostra o grafo para o exemplo do objeto o_8 de DEPARTAMENTO apresentado anteriormente.

O modelo anterior permite dois tipos de definição para igualdade na comparação dos estados de dois objetos. Afirma-se que dois objetos têm **estados idênticos** (igualdade profunda) se os grafos que representam seus estados forem idênticos em todos aspectos, incluindo os OID s em cada nível. Outra definição menos precisa de igualdade é no caso de dois objetos terem **estados iguais** (igualdade rasa). Nesse caso, a estrutura do grafo deve ser a mesma e todos os valores atômicos correspondentes no grafo também devem ser os mesmos. Porém, alguns nós internos correspondentes nos dois grafos podem possuir OID s diferentes.

EXEMPLO 2: Objetos idênticos versus objetos iguais

Um exemplo pode ilustrar a diferença entre as duas definições para comparação de igualdade entre estados de objetos. Considere os seguintes objetos O_1, O_2, O_3, O_4, O_5 e O_6 :

$O_1 = (i_1, \text{tuple}, \langle a_1:i_4, a_2:i_6 \rangle)$

$O_2 = (i_2, \text{tuple}, \langle a_1:i_5, a_2:i_6 \rangle)$

$O_3 = (i_3, \text{tuple}, \langle a_1:i_4, a_2:i_6 \rangle)$

$O_4 = (i_4, \text{atom}, 10 >)$

$O_5 = (i_5, \text{atom}, 10 >)$

$O_6 = (i_6, \text{atom}, 20 >)$

Os objetos O_1 e O_2 têm estados *iguais*, uma vez que seus estados no nível atômico são os mesmos, mas os valores são alçados por meio dos objetos diferentes O_4 e O_5 . Por outro lado, os estados dos objetos O_1 e O_3 são *idênticos*, embora os objetos O_1 e O_3 sejam, por possuírem OID s distintos. De modo semelhante, apesar de os estados de O_4 e O_5 serem idênticos, os objetos reais O_4 e O_5 são iguais, mas não idênticos, por possuírem OID s diferentes.

20.2.3 Construtores de Tipos

Uma **linguagem de definição de objeto (ODL — Object Definition Language)** que incorpora os construtores de tipos anteriores pode ser usada para definir tipos de objetos para determinada aplicação de banco de dados. No Capítulo 21, será descrito o padrão ODL da ODMG, mas inicialmente os conceitos serão introduzidos gradualmente nesta seção, utilizando uma notação mais simples. Os construtores de tipos podem ser utilizados para definir as *estruturas de dados* para um *esquema de banco de dados*.

16 Estes objetos atômicos são os que podem causar problemas, por causa do uso de muitos identificadores de objetos, caso esse modelo se implementado diretamente.

17 Corresponderia à DDL (Data Definition Language) do sistema de banco de dados (Capítulo 2).

20.2 Identidade de Objeto, Estrutura de Objeto e Construtores de Tipos

465

dos OO. Na Seção 20.3 será mostrado como incorporar a definição de *operações* (ou métodos) em um esquema OO. A Figura 20.2 mostra como podem ser declarados os tipos Empregado e Departamento correspondentes às instâncias de objetos mostradas na Figura 20.1. Na Figura 20.2 o tipo Data é definido como uma tupla em vez de um valor atômico como na Figura 20.1. São utilizadas as palavras-chave tuple, set e list para os construtores de tipos, e tipos de dados padrões são disponibilizados para tipos atômicos (integer, string, float, e assim por diante).

LEGENDA: Q objeto tuple set

FIGURA 20.1 Representação de um objeto complexo DEPARTAMENTO como um grafo.

Atributos que se referem a outros objetos — como depto de Empregado ou projetos de Departamento — são basicamente referências para outros objetos, servindo, assim, para representar *relacionamentos* entre os tipos de objetos. Por exemplo, o atributo depto de Empregado é do tipo Departamento, sendo assim utilizado para se referir a um objeto Departamento particular (onde Empregado trabalha). O valor de tal atributo seria um OID para um objeto Departamento específico. Um relacionamento binário pode ser representado em uma única direção ou pode possuir uma *referência inversa*. Essa representação permite percorrer o relacionamento facilmente em ambas as direções. Por exemplo, o atributo empregados de Departamento tem como seu valor um *conjunto de referências* (isto é, um conjunto de OIDs) para objetos do tipo Empregado; estes são os empregados que trabalham no departamento. O inverso é o atributo de referência depto de Empregado. Será visto no Capítulo

466

Capítulo 20 Conceitos para Bancos de Dados de Objetos

21 como o padrão ODMG permite que inversos sejam explicitamente declarados como atributos de relacionamentos para garantir que as referências inversas sejam coerentes.

```

define type Empregado
  tuple (
    pnome:          string;
    minicial        char;
    unome:          string;
    ssn:            string;
    datanasc:       Data;
    endereço:       string;
    sexo:           char;
    salário:        float;
    supervisor:     Empregado;
    depto:          Departamento;);

define type Data
  tuple (
    dia:            integer;
    mes:            integer;
    ano:            integer; );

define type Departamento
  tuple (
    dnome:          string;
    dnumero:        integer;
    ger:            tuple ( gerente:      Empregado;
                          ger_data_:    Data; );
    localizações:   set(string);
    empregados:     set(Empregado);
    projetos:       set(Projeto); );

```

FIGURA 20.2 Especificação dos tipos de objetos Empregado, Data e Departamento com o uso de construtores de tipos.

20.3 ENCAPSULAMENTO DE OPERAÇÕES, MÉTODOS E PERSISTÊNCIA

O conceito de *encapsulamento* é uma das principais características das linguagens e dos sistemas OO. Ele está relacionado também com os conceitos de *tipos abstratos de dados* e *ocultar a informação* nas linguagens de programação. Nos tradicionais modelos e sistemas de bancos de dados, esse conceito não é aplicado, uma vez que é costumeiro deixar a estrutura do banco de dados visível para os usuários e os programas externos. Nesses modelos convencionais, algumas operações padronizadas do banco de dados são aplicáveis a todos os tipos de objetos. Por exemplo, no modelo relacional, as operações para seleção, inserção, remoção e modificação de tuplas são genéricas e podem ser aplicadas a *qualquer relação* no banco de dados. A relação e seus atributos são visíveis para os usuários e programas externos que acessam a relação por meio do uso dessas operações.

20.3.1 Especificando o Comportamento do Objeto através de Operações de Classe

Os conceitos ocultar a informação e encapsulamento podem ser aplicados a objetos de bancos de dados. A idéia principal é definir o **comportamento** de um tipo de objeto com base nas **operações** que podem ser externamente aplicadas aos objetos daquele **tipo**. A estrutura interna do objeto é escondida e o objeto é acessível por meio de um número de operações predefinidas. Algumas operações podem ser utilizadas para criar (insert) ou destruir (delete) objetos; outras operações podem atualizar o estado do objeto e outras podem ser utilizadas para recuperar partes do estado do objeto ou para aplicar alguns cálculos. Outras operações ainda podem combinar recuperação, cálculos e atualização. Em geral, a **implementação** de uma operação pode ser especificada em uma *linguagem de programação de propósito geral* que oferece flexibilidade e capacidade para definir essas operações.

Os usuários externos de um objeto conhecem apenas a **interface** do tipo do objeto, o qual define o nome e os argumentos (parâmetros) de cada operação. A implementação é oculta para os usuários externos, o que inclui a definição das estruturas internas de dados do objeto e a implementação das operações que acessam essas estruturas. Na terminologia OO, a parte da interface de cada operação é chamada **assinatura**, enquanto a implementação da operação é chamada **método**. Normalmente um método é invocado pela passagem de uma **mensagem** ao objeto para que seja executado o método correspondente. Deve ser observado que, como parte da execução de um método, uma mensagem subsequente para outro objeto pode ser enviada e esse mecanismo pode ser utilizado para retomar valores a partir do ambiente externo ou para outros objetos.

20.3 Encapsulamento de Operações, Métodos e Persistência

467

Para aplicações de bancos de dados, o requisito de que todos os objetos sejam completamente encapsulados é muito rígido. Uma maneira de flexibilizar esse requisito é dividir a estrutura de um objeto em atributos (variáveis de instância) visíveis e **ocultos**. Os atributos visíveis podem ser acessados diretamente para leitura por operadores externos ou por uma linguagem de consulta de alto nível. Os atributos ocultos de um objeto são completamente encapsulados e podem ser acessados somente por meio de operações predefinidas. No Capítulo 21 será apresentada a linguagem de consulta OQL, que é proposta como uma linguagem de consulta padrão para BDOOs.

Na maioria dos casos, as operações que *atualizara* o estado de um objeto são encapsuladas. Esse é um modo de definir a semântica de atualização de objetos, uma vez que em muitos modelos de dados OO poucas restrições de integridade são predefinidas no esquema. Cada tipo de objeto tem suas restrições de integridade programadas *internamente nos métodos* que criam, removem e atualizam objetos, com código escrito de forma explícita para checar as violações de restrições e para manipular exceções. Em tais casos, todas as operações de atualização são implementadas por meio de operações encapsuladas. Mais recentemente, a linguagem ODL do padrão ODMG permitiu a especificação de algumas restrições comuns, tais como chaves e relacionamentos inversos (integridade referencial), de modo que o sistema pudesse garantir automaticamente essas restrições (Capítulo 21).

O termo **classe** é freqüentemente utilizado para se referir a uma definição do tipo objeto, junto com as definições das

operações para esse tipo. A Figura 20.3 mostra como as definições de tipos da Figura 20.2 podem ser estendidas com as operações para definir as classes. Algumas operações são declaradas para cada classe, e a assinatura (interface) de cada operação é acrescida à definição da classe. Um método (implementação) para cada operação deve ser definido em outro local utilizando uma linguagem de programação. As operações típicas incluem o **construtor** do objeto, utilizado para criar um novo objeto, e o **destrutor**, utilizado para destruir um objeto. Operações **modificadoras** de objetos também podem ser declaradas para modificar os estados (valores) dos vários atributos de um objeto. Operações adicionais podem **recuperar** informações sobre o objeto.

define class Empregado:

```

type tuple ( pnome:          string;
minicial:          char;
unome:            string;
ssn:              string;
datanasc:         Data;
endereço:         string;
sexo:             char;
salário:          float;
supervisor:       Empregado;
depto:            Departamento;
operations idade:    integer;
cria^emp:         Empregado;
excluLemp:        boolean;
end Employee;
```

define class Departamento:

```

type tuple ( dnome:          string;
dnumero:          integer;
ger:              tuple (      gerente: Empregado;
                             ger_data_inicio: Data

localizações:     set(string);
empregados:       set(Empregado);
projetos:         set(Projeto);
operations (nro_empregados: integer;
cria_depto:       Departamento;
exclui_depto:     boolean;
aloca_emp(e:Empregado): boolean; (* adiciona um empregado a um departamento *)
desaloca_emp(e:Empregado): boolean; (* exclui um empregado de um departamento *) end Department;
```

FIGURA 20.3 Adicionando operações às definições de Empregado e Departamento.

18 Esta definição de classe é semelhante àquela utilizada na conhecida linguagem de programação C++. O padrão ODMG utiliza a palavra *interface* adicionalmente à *classe* (Capítulo 21). No modelo EER o termo *classe* foi utilizado para se referir a um tipo de objeto, junto com o conjunto de todos os objetos desse tipo (Capítulo 4).

Capítulo 20 Conceitos para Bancos de Dados de Objetos

Uma operação normalmente é aplicada a um objeto utilizando a **notação ponto** (dot). Por exemplo, se *d* é uma referência a um objeto departamento, pode-se chamar uma operação como `numero_de_empregados` escrevendo `d.numero_de_empregados`. Analogamente, escrevendo `d.destroy_depto`, o objeto referenciado por *d* é destruído (removido). A única exceção é a operação construtor, à qual retorna uma referência a um novo objeto departamento. Assim, normalmente há um nome padrão para a operação construtor da classe, que é o próprio nome da classe, apesar de não ter sido utilizada na Figura 20.3. A notação de ponto é também utilizada para se referir aos atributos de um objeto — por exemplo, escrevendo `d.numero` ou `d.ger.datainicio`.

20.3.2 Especificando Persistência de Objeto através de Nomeação e Alcançabilidade

Um SGBDOO normalmente possui um acoplamento forte com uma LPOO, que é utilizada para especificar a implementação dos métodos, assim como outros códigos da aplicação. Normalmente um objeto é criado por algum programa de aplicação em execução, pela chamada da operação construtor do objeto. Nem todos os objetos criados são armazenados permanentemente no banco de dados. Os **objetos transientes** existem na execução do programa e desaparecem quando o programa termina. Os **objetos persistentes** são armazenados no banco de dados e persistem após o término do programa. Os mecanismos usuais para tornar um objeto persistente são a *nomeação* e a *alcançabilidade*.

O **mecanismo de nomeação** consiste em dar a um objeto um nome persistente único pelo qual ele possa ser recuperado pelo programa em execução e outros programas. Este nome de objeto persistente pode ser atribuído por meio de um comando específico ou uma operação no programa, como mostrado na Figura 20.4. Todos os nomes atribuídos aos objetos devem ser únicos dentro de um determinado banco de dados. Assim, os objetos persistentes nomeados são utilizados como **pontos de entrada** pelos quais os usuários e as aplicações podem iniciar o acesso ao banco de dados. Obviamente, não é prático dar nomes a todos os objetos em um grande banco de dados, que inclui milhares de objetos, de forma que a maioria deles se torna persistente por meio de um segundo mecanismo, chamado **alcançabilidade**. Este mecanismo torna o objeto alcançável a partir de algum objeto persistente. Um objeto *B* é dito **alcançável** a partir de um objeto *A* se uma sequência de referências no grafo conduz, a partir do objeto *A*, ao objeto *B*. Como exemplo, todos os objetos na Figura 20.1 são alcançáveis a partir do objeto *os*, portanto, se *os* se tornar persistente, todos os objetos na Figura 20.1 também virão a se tornar persistentes.

Se inicialmente for criado um objeto persistente chamado *N*, cujo estado é um *conjunto (set)* ou uma *lista (list)* de objetos de uma classe *C*, os objetos de *C* podem se tornar persistentes *adicionando-os* ao conjunto ou à lista, tornando-os assim alcançáveis a partir de *N*. Desse modo, *N* define uma **coleção persistente** de objetos da classe *C*. Por exemplo, pode-se definir a classe `Conjunto_de_Departamentos` (Figura 20.4) cujos objetos são do tipo `set(Departamento)`. Suponha que um objeto do tipo `Conjunto_de_Departamentos` é criado e que seja nomeado `Todos_Departamentos`, tornando-se dessa forma persistente, como ilustrado na Figura 20.4. Qualquer objeto `Departamento` que é adicionado ao conjunto `Todos_Departamentos` pelo uso da operação `adiciona_depto` torna-se persistente em virtude de ser alcançável por `Todos_Departamentos`. O objeto `Todos_Departamentos` é normalmente chamado **extensão** da classe `Departamento`, uma vez que manterá todos os objetos persistentes do tipo `Departamento`. Conforme será visto no Capítulo 21, o padrão ODL do ODMG fornece ao projetista de esquemas a opção de identificar uma extensão como parte da definição de uma classe.

Deve ser observada a diferença entre os modelos de banco de dados convencionais e bancos de dados *OO*. Em modelos de bancos de dados convencionais, tais como o modelo relacional ou o modelo EER (Entidade Relacionamento Estendido), assume-se que *todos* os objetos são persistentes. Assim, quando um tipo de entidade ou classe como `EMPREGADO` é definido no modelo EER, ele representa tanto a *declaração de tipo* para `EMPREGADO` quanto um *conjunto persistente* de *todos* os objetos `EMPREGADO`. Na abordagem *OO*, uma declaração de classe para `EMPREGADO` especifica somente o tipo e as operações para uma classe de objetos. O usuário deve definir separadamente um objeto persistente do tipo `set(EMPREGADO)` ou `list(EMPREGADO)`, cujo valor é a *coleção de referências* para todos os objetos persistentes `EMPREGADO`, se for desejado, como ilustrado na Figura 20.4. Isso permite que objetos transientes e persistentes sigam o mesmo tipo e declarações de classe da ODL e da LPOO, respectivamente. Em geral, é possível definir várias coleções persistentes para a mesma definição de classe, se desejável.

19 Nomes padrões para as operações construtor e destrutor existem na linguagem de programação C++ . Por exemplo, para a classe `Empregado` o nome padrão para o construtor é `Empregado` e o nome padrão para o destrutor é `-Empregado`. Também é comum utilizar a operação *neui* para criar novos objetos.

20 Como será visto no Capítulo 21, a sintaxe ODL da ODMG utiliza `set<Departamento>` em vez de `set(Departamento)`.

21 Alguns sistemas, como o POET, criam automaticamente a extensão para uma classe.

20.4 Hierarquias de Classe e Tipo, e Herança

469

```

define class Conjunto_de_Departamentos: type          set(Departamento);
operations  adicionaDepto(d:Departamento):  a Dboolean:
(* adiciona um departamento de um objeto Conjunto_de_Departamentos *)
removeDepto(d:Departamento):boolean; (* exclui um departamento de um objeto Conjunto_de_Departamentos *)
cria_Conjunto_de_Departamentos:      Conjunto_de_Departamentos; destroLConjunto_de_Departamentos:  boolean; end
Conjunto_de_Departamentos;
persistem name TodosDepartamentos: Conjunto_de_Departamentos;
(* TodosDepartamentos é um objeto persistente nomeado do tipo Conjunto_de_Departamentos *)
d:= cria_Depto;
(* cria um novo objeto departamento na variável d *)
b:= TodosDepartamentos. adicionaJDepto(d);
(* torna d persistente pela adição do mesmo ao conjunto persistente TodosDepartamentos *)

```

FIGURA 20.4 Criando objetos persistentes por nomeação e alcançabilidade.

20.4 HIERARQUIAS DE CLASSE E TIPO, E HERANÇA

Uma outra característica essencial em sistemas de bancos de dados OO é que eles permitem hierarquias de tipo e herança. As hierarquias de tipo em bancos de dados normalmente acarretam uma restrição nas extensões correspondentes aos tipos na hierarquia. Inicialmente serão discutidas as hierarquias de tipo (Seção 20.4.1), em seguida, as restrições sobre as extensões (Seção 20.4.2). Será utilizado um modelo OO diferente nesta seção — um modelo no qual atributos e operações são tratados de modo uniforme —, uma vez que tanto atributos quanto operações podem ser herdados. No Capítulo 21, será discutido o modelo de herança no padrão ODMG, o qual difere do modelo aqui discutido.

20.4.1 Hierarquias de Tipo e Herança

Na maioria das aplicações de bancos de dados há uma grande quantidade de objetos do mesmo tipo ou classe. Assim, os bancos de dados OO devem ser capazes de classificar objetos com base em seus tipos, como fazem outros sistemas de bancos de dados. Mas em bancos de dados OO um requisito adicional é que o sistema permita a definição de novos tipos baseados em outros tipos predefinidos, formando uma **hierarquia de tipos** (ou de classes).

Normalmente um tipo é definido pela atribuição de um nome e então se definem seus atributos (variáveis de instância) e operações (métodos) para esse tipo. Em alguns casos, os atributos e as operações são conjuntamente chamados *funções*, uma vez que os atributos são semelhantes a funções com zero argumento. Um nome de função pode ser utilizado para se referir ao valor de um atributo ou ao valor resultante de uma operação (método). Nesta seção será utilizado o termo **função** tanto para atributos *quanto* para operações de um tipo objeto porque serão tratados de modo semelhante em uma introdução básica a herança.

Um tipo, em sua forma mais simples, pode ser definido pela atribuição de um **nome do tipo** e depois listando os nomes de suas funções visíveis (*públicas*). Quando se especificar um tipo nesta seção, será utilizado o seguinte formato, o qual não especifica os argumentos das funções:

NOME_DO_TIPO: funcao, funcao..... funcao

22 Nesta seção serão utilizados os termos *tipo* e classe para significar a mesma coisa — os atributos e as operações de um tipo de objeto.

23 Será visto no Capítulo 21 que tipos com funções são semelhantes às interfaces utilizadas na ODL do ODMG.

Por exemplo, um tipo que descreve as características de uma PESSOA pode ser definido como: PESSOA: Nome, Endereço, Data_de_nascimento, Idade, SSN

No tipo PESSOA, Nome, Endereço, SSN e Data_de_Nascimento podem ser implementados como atributos armazenados, enquanto a função Idade pode ser implementada como um método que calcula a idade a partir do atributo Data_de_Nascimento e da data atual. O conceito de subtipo é útil quando o projetista ou o usuário precisa criar um novo tipo semelhante mas não idêntico a um tipo já definido. O subtipo herda, então, todas as funções do tipo predefinido, que podemos chamar de supertipo. Como exemplo, supondo que se deseja definir dois novos tipos EMPREGADO e ESTUDANTE:

EMPREGADO: Nome, Endereço, Data_de_Nascimento, Idade, SSN, Salário, DataContratacao, Experiência ESTUDANTE: Nome, Endereço, Data_de_Nascimento, Idade, SSN, Maior, MPG

Uma vez que tanto ESTUDANTE quanto EMPREGADO incluem todas as funções definidas para PESSOA mais algumas funções adicionais que lhe são próprias, podemos então declará-las como subtipos de PESSOA. Cada uma herdará as funções anteriormente definidas em PESSOA — Nome, Endereço, Data_de_Nascimento, Idade e SSN. Para ESTUDANTE, é necessário somente definir as novas funções (locais) Especialização e MPG (Médias Pontos Graduação), que não são herdadas. Provavelmente, pode-se definir Especialização como um atributo armazenado, enquanto MPG pode ser implementado como um método que calcula o valor médio das notas do estudante pelo acesso aos valores que estão armazenados internamente (ocultos) em cada objeto ESTUDANTE como *atributos privados*. Para EMPREGADO, as funções salário e DataAdmissão podem ser atributos armazenados, enquanto Experiência pode ser um método que calcula a experiência a partir do valor da data de admissão.

A idéia da definição de um tipo envolve definir todas as suas funções e implementá-las como atributos ou como métodos. Quando um subtipo é definido, ele pode herdar todas essas funções e suas implementações. Somente as funções que são específicas ou locais ao subtipo e que, portanto, não foram especificadas no supertipo, precisam ser definidas e implementadas. Assim, podem ser declarados EMPREGADO e ESTUDANTE da seguinte forma:

EMPREGADO **subtype-of** PESSOA: Salário, DataAdmissão, Experiência ESTUDANTE **subtype-of** PESSOA: Especialização, MPG

Em geral, um subtipo inclui todas as funções que estão definidas para seu supertipo, mais algumas funções adicionais que são específicas somente para o subtipo. Assim, é possível gerar uma hierarquia de tipos para mostrar os relacionamentos supertipo/subtipo entre todos os tipos declarados no sistema.

Como outro exemplo, considere um tipo que descreve objetos geométricos planos que podem ser definidos como:

OBJETO_GEOMETRICO: Formato, Área, Ponto_de_Referencia

Para o tipo OBJETO_GEOMETRICO, Formato é implementado como um atributo (seu domínio pode ser um tipo enumerado com os valores 'triângulo', 'retângulo', 'círculo', e assim por diante) e Área como um método que é aplicado para calcular a área.

Ponto_de_Referencia especifica as coordenadas de um ponto que determina a localização do objeto. Agora, suponha que se deseja definir alguns subtipos para o tipo OBJETO_GEOMETRICO, como segue:

RETÂNGULO **subtype-of** OBJETO_GEOMETRICO: Largura, Altura TRIÂNGULO **subtype-of** OBJETO_GEOMETRICO: Lado1, Lado2, Ângulo CIRCULO **subtype-of** OBJETO_GEOMETRICO: Raio

Observe que a operação Área pode ser implementada por um método diferente para cada subtipo, uma vez que o procedimento para o cálculo da área é diferente para retângulos, triângulos e círculos. De modo similar, o atributo Ponto_de_Referencia pode ter um significado diferente para cada subtipo: pode ser o ponto central para objetos de RETÂNGULO e CIRCULO e o ponto de vértice entre os dois lados de um objeto TRIÂNGULO. Alguns sistemas de bancos de dados OO permitem a renomeação de funções herdadas pelos diferentes subtipos para refletir mais fortemente seu significado.

Uma maneira alternativa de se declarar esses três subtipos é especificar o valor do atributo Formato como uma condição que deve ser satisfeita para objetos de cada subtipo:

RETÂNGULO **subtype-of** OBJETO_GEOMETRICO (Formato='retângulo'): Largura, Altura TRIÂNGULO **subtype-of** OBJETO_GEOMETRICO: (Formato='triângulo'): Lado1, Lado2, Ângulo CIRCULO **subtype-of** OBJETO_GEOMETRICO: (Formato='círculo'): Raio

Aqui, somente objetos OBJETO_GEOMETRICO, nos quais Formato='retângulo', são do subtipo RETÂNGULO e, de modo similar, os outros dois subtipos. Nesse caso, todas as funções do supertipo OBJETO_GEOMETRICO são herdadas usando-se cada um dos três subtipos, mas o valor do atributo Formato é restrito a um valor específico para cada um deles.

20.5 Objetos Complexos 471

Observe que a definição de tipos descreve objetos, mas *não* gera objetos por si mesma. São apenas declarações de determinados tipos; e, como parte da declaração, é especificada a implementação das funções de cada tipo. Em uma aplicação de banco de dados, há muitos objetos de cada tipo. Quando um objeto é criado, geralmente pertence a um ou mais desses tipos que foram declarados. Por exemplo, um objeto círculo é dos tipos CIRCULO e OBJETO_GEOMETRICO (por herança). Cada objeto também se torna um membro de uma ou mais coleções persistentes (ou extensões), às quais são utilizadas para agrupar coleções de objetos que têm o mesmo significado na aplicação de banco de dados.

20.4.2 Restrições em Extensões Correspondentes a uma Hierarquia de Tipos

Na maioria dos bancos de dados OO, a coleção de objetos em uma extensão possui o mesmo tipo ou classe. Entretanto, essa não é uma condição necessária. Por exemplo, a SMALLTALK, uma linguagem OO chamada *sem tipos*, permite que uma coleção de objetos contenha objetos de diferentes tipos. Esse também pode ser o caso quando outras linguagens sem tipo, não orientadas a objetos, como LISP, são estendidas com conceitos OO. No entanto, uma vez que a maioria dos bancos de dados suporta tipos, até o fim desta seção assumiremos que **extensões** são coleções de objetos do mesmo tipo.

É comum em aplicações de bancos de dados que cada tipo ou subtipo possua uma extensão associada, que mantenha a coleção de todos os objetos persistentes daquele tipo ou subtipo. Nesse caso, a restrição é de que todo objeto numa extensão que corresponda a um subtipo também deva ser um membro da *extensão* que corresponda a seu supertipo. Alguns sistemas de bancos de dados OO possuem um tipo de sistema predefinido (chamado de classe RAIZ (ROOT) OU classe OBJETO), cuja extensão contém todos os objetos do sistema. A classificação então segue, designando objetos para subtipos adicionais que são significativos para a aplicação, criando uma **hierarquia de tipo** ou **hierarquia de classe** para o sistema. Todas as extensões para classes definidas pelo sistema e pelo usuário são subconjuntos da extensão correspondente à classe OBJETO, direta ou indiretamente. No modelo ODMG (Capítulo 21) o usuário pode ou não especificar uma extensão para cada classe (tipo) dependendo da aplicação.

Na maioria dos sistemas OO, é feita uma distinção entre objetos persistentes e transientes e coleções. Uma **coleção persistente** mantém uma coleção de objetos que é armazenada permanentemente no banco de dados e pode dessa forma ser acessada e compartilhada por diferentes programas. Uma **coleção transiente** existe temporariamente durante a execução de um programa, mas não é mantida quando o programa termina. Por exemplo, uma coleção transiente pode ser criada num programa para manter o resultado de uma consulta que seleciona alguns objetos em uma coleção persistente e copia aqueles objetos para uma coleção transiente. A coleção transiente mantém o mesmo tipo de objetos da coleção persistente. O programa pode, dessa forma, manipular os objetos na classe transiente e, uma vez terminado o programa, a coleção transiente deixa de existir. Em geral, inúmeras coleções — transientes ou persistentes — podem conter objetos do mesmo tipo.

Note que os construtores de tipos discutidos na Seção 20.2 permitem que o estado de um objeto seja uma coleção de objetos. Dessa forma, objetos de coleções cujos tipos sejam baseados em um *construtor de conjunto* podem definir várias coleções — uma correspondente a cada objeto. Os objetos valorados para o conjunto são, eles próprios, membros de uma outra coleção. Isso possibilita esquemas de classificação multinível, em que um objeto em uma coleção possui, como seu estado, uma coleção de objetos de uma classe diferente.

Como será visto no Capítulo 21, o modelo ODMG distingue entre herança de tipo — chamada herança de interface e representada pelo símbolo ":" — e restrição de herança de extensão — representada pela palavra-chave EXTEND.

20.5 OBJETOS COMPLEXOS

Uma das principais motivações que levaram ao desenvolvimento de sistemas OO foi o desejo de representar objetos complexos. Existem dois tipos principais de objetos complexos: estruturados e não estruturados. Um objeto complexo estruturado é constituído de componentes e é definido pela aplicação de construtores de tipos disponíveis, de modo recursivo, em vários níveis. Um objeto complexo não estruturado, em geral, é um tipo de dado que requer um grande volume de armazenamento, tal como um tipo de dado que representa uma imagem ou um objeto de texto longo.

20.5.1 Objetos Complexos Não Estruturados e Extensibilidade de Tipo

Uma vantagem de lidar com **objetos complexos não estruturados** oferecida por um SGBD é que eles permitem o armazenamento e a recuperação de grandes objetos que são necessários à aplicação do banco de dados. Exemplos típicos desses objetos são *imagens bitmap* e strings *{cadeias de caracteres} de texto longo* (como documentos); também são conhecidos como **binary large objects** (**objetos binários extensos**) ou **BLOBs**. Cadeias de caracteres também são conhecidas como **objetos de textos ex-**

24 Este é o chamado OBJECT no modelo ODMG (Capítulo 21).

Capítulo 20 Conceitos para Bancos de Dados de Objetos

tenso (*character large objects*), ou CLOBs. Esses objetos são não estruturados, o que significa que o SGBD não conhece sua estrutura — somente a aplicação que os utiliza pode interpretar seu significado. Por exemplo, a aplicação pode possuir funções para exibir uma imagem ou procurar algumas palavras-chave numa string de texto longo. Os objetos são considerados complexos porque exigem uma grande área de armazenamento e não são parte dos tipos de dados padrões fornecidos pelos SGBDs convencionais. Uma vez que o objeto é muito grande, o SGBD pode recuperar uma parte do objeto e fornecê-lo ao programa de aplicação antes que o objeto seja recuperado por inteiro. O SGBD também pode utilizar técnicas de *buffering* e *cache* para buscar previamente partes do objeto antes que o programa de aplicação precise acessá-los.

Os softwares do SGBD não possuem capacidade para processar diretamente condições de seleção e outras operações realizadas em valores desses objetos, a não ser que a aplicação forneça o código para realizar as operações de comparação necessárias para a seleção. Em um SGBDOO, isso pode ser obtido definindo-se o novo tipo de dado abstrato para os objetos não interpretados e fornecendo os métodos para seleção, comparação e apresentação desses objetos. Por exemplo, considere objetos que são imagens *bitmap* bidimensionais. Suponha que a aplicação precise selecionar a partir de uma coleção de tais objetos somente aqueles que incluam certo padrão. Nesse caso, o usuário deve fornecer o programa de reconhecimento do padrão, como um método em objetos do tipo *bitmap*. O SGBDOO recupera, então, um objeto do banco de dados e aplica nele método para reconhecimento do padrão para determinar se o objeto adere ao padrão desejado.

Uma vez que um SGBDOO permite aos usuários a criação de novos tipos, e tendo em vista que um tipo inclui tanto a estrutura quanto as operações, podemos visualizar um SGBDOO como tendo um sistema de tipos extensíveis. Podem-se criar bibliotecas de novos tipos, definindo suas estruturas e operações, incluindo tipos complexos. As aplicações podem então utilizar ou modificar esses tipos, no último caso criando subtipos dos tipos fornecidos nas bibliotecas. Entretanto, o SGI deve possibilitar o armazenamento principal e capacidades de recuperação para objetos que necessitam de grandes volumes de armazenamento, de modo que as operações possam ser aplicadas de modo eficiente. Muitos SGBDOOs estão preparados para armazenamento e recuperação de grandes objetos não estruturados, como strings de caracteres ou strings de bits, que podem ser devolvidos 'as-is' (como estão) ao programa de aplicação para interpretação. Recentemente, os SGBDs relacionais relacionais também têm sido capazes de oferecer essas funcionalidades. Também têm sido desenvolvidas técnicas especiais de indexação.

20.5.2 Objetos Complexos Estruturados

Um objeto complexo estruturado diferencia-se de um objeto complexo não estruturado no sentido de que a estrutura do objeto é definida pela aplicação recursiva dos construtores de tipo oferecidos pelo SGBDOO. Assim, a estrutura do objeto é definida e conhecida pelo SGBDOO. Como um exemplo, considere o objeto DEPARTAMENTO mostrado na Figura 20.1. No primeiro nível, o objeto possui uma estrutura de tupla com seis atributos: *NOME*, *NUMEROD*, *GER*, *LOCALIZAÇÕES*, *EMPREGADOS* e *PROJETOS*. Contudo apenas dois desses atributos — identificados como *NOME* e *NUMEROD* — possuem valores básicos; os outros quatro possuem uma estrutura complexa e, desse modo, foi construído o segundo nível da estrutura do objeto complexo. Um desses quatro (*GER* tem uma estrutura de tupla, e os outros três (*LOCALIZAÇÕES*, *EMPREGADOS* e *PROJETOS*) possuem estruturas de conjuntos. No terceiro nível, para um valor de tupla de *GER*, temos um atributo básico (*GERDATAINICIO*) e um atributo (*GERENTE*) que se refere a um objeto empregado, o qual possui uma estrutura de tupla. Para um conjunto *LOCALIZAÇÕES*, temos um conjunto de valores básicos, mas para ambos os conjuntos *EMPREGADOS* e *PROJETOS* temos conjuntos de objetos estruturados como tuplas. Há dois tipos de semânticas de referência entre um objeto complexo e seus componentes em cada nível. O primeiro tipo, o qual podemos chamar de semântica de propriedade, trata-se do caso de subobjetos de um objeto complexo serem encapsulados dentro do objeto complexo, sendo assim considerados como partes do objeto complexo. O segundo tipo, o qual podemos chamar de semântica de referência, trata-se do caso de componentes do objeto complexo serem, por si, objetos independentes, mas poderem ser referenciados a partir do objeto complexo. Por exemplo, podemos considerar os atributos *NOME*, *NUMEROD*, *GER* e *LOCALIZAÇÕES* como sendo pertencentes a um DEPARTAMENTO, enquanto *EMPREGADOS* e *PROJETOS* são referências porque SE referenciam como objetos independentes. O primeiro tipo também é conhecido como relacionamento 'é-parte-de' ou 'é-componente-de', enquanto o segundo tipo é chamado como relacionamento 'é-associado-com', uma vez que descreve uma associação no mesmo nível entre dois objetos independentes. O relacionamento 'é-parte-de' (semântica de propriedade) para construção de objetos complexos tem a propriedade de os componentes do objeto serem encapsulados dentro do objeto complexo e serem considerados como parte de seu estado interno. Eles não necessitam de identificadores de objetos e somente podem ser acessados por métodos do objeto proprietário. São removidos quando o objeto proprietário é removido. Por outro lado, componentes referenciados são considerados como objetos independentes que podem possuir suas próprias identidades e métodos. Quando um objeto complexo necessita acessar seus componentes referenciados, ele deve fazê-lo por meio da chamada de métodos apropriados de seus componentes, uma vez que eles não estão encapsulados no objeto complexo.

20.6 Outros Conceitos Orientados a Objetos

473

xo. Assim, a semântica de referência representa *relacionamentos* entre objetos independentes. Além disso, um objeto componente referenciado pode ser referenciado por mais de um objeto complexo, não sendo automaticamente removido quando o objeto complexo é apagado.

Um SGBDOO deve oferecer opções de armazenamento para agrupar (clustering) objetos componentes de um objeto complexo em memória secundária com o objetivo de aumentar a eficiência das operações que acessam o objeto complexo. Em muitos casos, a estrutura do objeto é armazenada em páginas de disco de forma não interpretada. Quando uma página de disco que inclui um objeto é recuperada para a memória, o SGBDOO pode montar o objeto complexo estruturado a partir das informações nas páginas do disco, as quais podem fazer referências a páginas adicionais no disco que devem ser recuperadas. Isso é conhecido como **montagem do objeto** complexo.

20.6 OUTROS CONCEITOS ORIENTADOS A OBJETOS

Nesta seção, apresentaremos uma visão geral de alguns outros conceitos OO, incluindo polimorfismo (sobrecarga de operador), herança múltipla, herança seletiva, versões e configurações.

20.6.1 Polimorfismo (Sobrecarga de Operador)

Uma outra característica de sistemas OO é que eles dão suporte ao **polimorfismo** de operações, que também é conhecido como **sobrecarga de operador**. Este conceito permite que o mesmo *nome de operador* ou *símbolo* seja associado a duas ou mais *implementações* diferentes para o operador, dependendo do tipo de objetos aos quais o operador é aplicado. Um exemplo simples de linguagens de programação pode ilustrar este conceito. Em algumas linguagens, o símbolo de operador '+' pode ter significados diferentes quando aplicado a operandos (objetos) de tipos diferentes. Se os operandos de '+' forem do tipo *inteiro*, a operação aplicada será uma adição inteira. Se os operandos '+' forem do tipo ponto *flutuante*, a operação aplicada será a adição em ponto flutuante. Se os operandos de '+' forem do tipo *conjunto*, a operação aplicada será a união de conjuntos. O compilador é capaz de determinar qual operação executar com base nos tipos de operandos fornecidos.

Em bancos de dados OO, uma situação semelhante ocorre. Pode ser utilizado o exemplo de OBJETCLGEOMETRICO, discutido na Seção 20.4 para ilustrar polimorfismo em bancos de dados OO. Suponha que OBJETCLGEOMETRICO e seus subtipos sejam declarados da seguinte forma:

```
OBJETO_GEOMETRICO: Formato, Área, Ponto_de_Referencia
RETANGULO subtype-of OBJETO_GEOMETRICO (Formato='retangulo'): Largura, Altura TRIÂNGULO subtype-of
OBJETO_GEOMETRICO (Formato='triângulo'): Lado1, Lado2, Angulo CIRCULO subtype-of OBJETO_GEOMETRICO
(Formato='círculo'): Raio
```

Aqui, a função Área é declarada para todos os objetos do tipo OBJETO_GEOMETRICO. Contudo, a implementação do método para Área pode ser diferente para cada subtipo de OBJETO_GEOMETRICO. Uma possibilidade é existir uma implementação geral para calcular a área de um OBJETO_GEOMETRICO genérico (escrevendo, por exemplo, um algoritmo para calcular a área de um polígono) e, em seguida, reescrever algoritmos mais eficientes para calcular as áreas de tipos específicos de objetos geométricos, como um círculo, um retângulo, um triângulo, e assim por diante. Nesse caso, a função Área é sobrecarregada por diferentes implementações.

O SGBDOO precisa selecionar, assim, o método adequado para a função Área com base no tipo do objeto geométrico ao qual é aplicado. Em sistemas fortemente tipados, isso pode ser realizado em tempo de compilação, uma vez que os tipos de objetos precisam ser conhecidos. Este caso é chamado **acoplamento antecipado ou estático (early (static) binding)**. Por outro lado, em sistemas com tipagem fraca ou inexistente (como SMALLTALK e LISP), o tipo do objeto ao qual a função é aplicada pode ser desconhecido até o momento da execução. Nesse caso, a função deve verificar o tipo do objeto no momento da execução e então chamar o método apropriado. Esse caso é chamado **acoplamento tardio ou dinâmico (late (dynamic) binding)**.

20.6.2 Herança Múltipla e Herança Seletiva

Em uma hierarquia de tipos, a **herança múltipla** ocorre quando um subtipo T é subtipo de dois (ou mais) tipos e, assim, herda as funções (atributos e métodos) de ambos os supertipos. Como exemplo, pode-se criar um subtipo ENGENHEIRO JERENTE, que é um subtipo tanto de GERENTE quanto de ENGENHEIRO. Isso leva à criação de um **reticulado de tipos** em vez de uma hierarquia de tipos. Um problema que pode ocorrer com herança múltipla é que os supertipos dos quais o subtipo herda podem possuir diferentes

25 Em linguagens de programação, há vários tipos de polimorfismo. Ao leitor interessado, os trabalhos são referenciados nas notas bibliográficas, que incluem uma discussão mais detalhada.

Capítulo 20 Conceitos para Bancos de Dados de Objetos

funções com o mesmo identificador, criando uma ambigüidade. Por exemplo, GERENTE e ENGENHEIRO podem ter uma função chamada Salário. Se a função Salário for implementada por diferentes métodos nos supertipos GERENTE e ENGENHEIRO, existe uma ambigüidade de qual das duas funções é herdada pelo subtipo ENGENHEIRO_GERENTE. É possível, entretanto, que tanto ENGENHEIRO quanto GERENTE herdem Salário de um mesmo supertipo (tal como EMPREGADO), um nível acima no reticulado. A regra geral é que se uma função for herdada de algum supertipo comum, então é herdada somente uma vez. Em tal caso, não há nenhuma ambigüidade; o problema surge somente se as funções são distintas em dois supertipos.

Há várias técnicas para lidar com ambigüidade em herança múltipla. Uma solução é executar uma verificação do sistema para ver se há ambigüidade quando o subtipo for criado, deixando o usuário escolher, explicitamente, qual função deve ser herdada naquele momento. Outra solução é utilizar algum default do sistema. Uma terceira solução é desabilitar herança múltipla se ocorrer ambigüidade de nome, forçando o usuário a mudar o nome de uma das funções em um dos supertipos. Alguns sistemas OO não permitem herança múltipla de modo nenhum.

A herança seletiva ocorre quando um subtipo herda somente algumas das funções de um supertipo, não herdando outras. Nesse caso, uma cláusula EXCEPT pode ser utilizada para listar as funções de um supertipo que não devem ser herdadas pelo subtipo.

Normalmente, o mecanismo de herança seletiva não é disponibilizado em sistemas de bancos de dados OO, mas é utilizado mais freqüentemente em aplicações de inteligência artificial.

20.6.3 Versões (Versionamento) e Configurações

Muitas aplicações de bancos de dados que utilizam sistemas OO exigem várias versões do mesmo objeto. Por exemplo, considere uma aplicação de banco de dados para um ambiente de engenharia de software, que armazena diversos tipos de software, como *módulos de projeto*, *módulos de código-fonte* e *informações de configuração*, para descrever quais módulos devem ser agregados para formar programas complexos e *casos de testes* para testar o sistema. Geralmente, são aplicadas *atividades de manutenção* a um sistema de software, conforme seus requisitos evoluem. A manutenção em geral acarreta modificações em alguns dos módulos de projeto e implementação. Se o sistema já é operacional e um ou mais módulos devem ser modificados, o projetista deve criar uma nova versão para cada um desses módulos para implementar as mudanças. Analogamente, novas versões de casos de testes devem ser geradas para testar essas novas versões dos módulos. Porém as versões existentes não devem ser descartadas até que as novas versões tenham sido completamente testadas e aprovadas; somente então as novas versões devem substituir as anteriores.

Observe que pode haver mais de duas versões de um mesmo objeto. Por exemplo, considere que dois programadores trabalham para atualizar o mesmo módulo de software simultaneamente. Nesse caso, duas versões, além do módulo original, são necessárias. Os programadores podem atualizar suas versões ao mesmo tempo, esse processo, é usualmente chamado engenharia concorrente.

Porém, eventualmente se torna necessário fundir essas duas versões tal que a nova versão (híbrida) possa incluir as modificações realizadas por ambos os programadores. Durante a fusão, também é necessário certificar-se de que as mudanças realizadas sejam compatíveis, o que acarreta, ainda, a criação de uma outra versão do objeto: aquela que é o resultado da fusão (merge) de duas versões atualizadas independentemente.

Como pode ser visto nessa discussão, um SGBDOO deve ser capaz de armazenar e gerenciar múltiplas versões do mesmo objeto conceituai. Vários sistemas suportam essa capacidade, permitindo que a aplicação mantenha diversas versões de um mesmo objeto e referencie-se explicitamente a determinadas versões quando necessário. No entanto, o problema de agrupar e reconciliar modificações realizadas em duas versões distintas normalmente é atribuído aos desenvolvedores da aplicação, que reconhecem sua semântica. Alguns SGBDs possuem certas facilidades para comparar duas versões do objeto original e determinar quando alguma das modificações são incompatíveis, com o objetivo de auxiliar o processo de 'merging' (fusão). Outros sistemas mantêm um grafo de versão que mostra os relacionamentos entre as versões. Sempre que uma versão v^A se originar pela cópia de uma outra versão v , um arco direcionado é desenhado de v até Vj . Analogamente, se duas versões v_2 e v_3 são fundidas, criando uma nova versão, v^A , são desenhados arcos direcionados a partir de v_2 e v_3 para v_4 . O grafo de versão pode auxiliar os usuários a compreender os relacionamentos entre as várias versões e podem ser utilizados internamente pelo sistema para gerenciar a criação e a remoção de versões.

Quando a criação de versões é aplicada a objetos complexos, surgem outras questões que devem ser tratadas. Um objeto complexo, como um sistema de software, pode ser constituído de muitos módulos. Quando é permitida a criação de versões, cada um desses módulos pode possuir uma série de versões distintas e seu próprio grafo de versão. Uma configuração de um objeto complexo é uma coleção formada por uma versão de cada módulo, de tal forma que as versões dos módulos sejam com-

26 No modelo ODMG, a herança de tipos se refere somente à herança de operações, não de atributos (Capítulo 21).

27 O problema de versões não é exclusivo de BDOOs, podendo ocorrer em SGBDs relacionais ou em outros.

20.7 Resumo 475

patíveis e que conjuntamente formem uma versão válida do objeto complexo. Uma nova versão ou configuração do objeto complexo não deve incluir necessariamente novas versões para todos os módulos. Assim, algumas versões de módulos que não tenham sido modificadas podem pertencer a mais de uma configuração do objeto complexo. Deve ser notado que uma configuração é uma coleção de versões de objetos *distintos* que, juntos, formam um objeto complexo, enquanto o grafo de versão descreve as versões do *mesmo* objeto. Uma configuração deve seguir a estrutura do tipo do objeto complexo; múltiplas configurações do mesmo objeto complexo são análogas a múltiplas versões de um objeto componente.

20.7 RESUMO

Neste capítulo foram discutidos os conceitos da abordagem orientada a objetos para sistemas de bancos de dados, os quais foram propostos para atender às necessidades de aplicações complexas de bancos de dados e para adicionar as funcionalidades de bancos de dados às linguagens de programação orientadas a objetos, como C++. Inicialmente foram discutidos os principais conceitos utilizados em bancos de dados OO, que incluem:

- *Identidade de objeto*: Os objetos possuem identificadores únicos que são independentes de seus valores de atributos.
- *Construtores de tipos*: Estruturas de objetos complexos podem ser construídas recursivamente aplicando-se um conjunto de construtores básicos, como tuple, set, list e bag.
- *Encapsulamento de operações*: A estrutura do objeto e as operações que podem ser aplicadas aos objetos são incluídas nas definições da classe de objetos.
- *Compatibilidade com a linguagem de programação*: Objetos transientes e persistentes são manipulados de modo uniforme. Os objetos tornam-se persistentes ao serem anexados a uma coleção persistente ou por nomeação explícita.
- *Hierarquias de tipo e herança*: Tipos objetos podem ser especificados pelo uso de uma hierarquia de tipos, a qual permite a herança de atributos e métodos de tipos anteriormente definidos. A herança múltipla é permitida em alguns modelos.
- *Extensões*: Todos os objetos persistentes de um tipo particular podem ser armazenados em uma extensão. As extensões correspondentes a uma hierarquia de tipos possuem restrições de conjunto/subconjunto associadas a elas.
- *Suporte a objetos complexos*: Objetos complexos estruturados e não estruturados podem ser armazenados e manipulados.
- *Polimorfismo e sobrecarga de operador*: Operações e nomes de métodos podem ser sobrecarregados para serem aplicados a diferentes tipos de objetos com diferentes implementações.
- *Criação de versões*: Alguns sistemas OO oferecem suporte para manutenção de diversas versões do mesmo objeto.

No próximo capítulo, será mostrado como alguns desses conceitos são tratados no padrão ODMG.

Questões de Revisão

- 20.1 Quais são as origens da abordagem orientada a objetos?
- 20.2 Quais características primárias deve possuir um OID?
- 20.3 Discuta sobre os vários construtores de tipos. Como eles são utilizados para criar estruturas de objetos complexos?
- 20.4 Discuta sobre o conceito de encapsulamento e diga como ele é utilizado para criar tipos abstratos de dados.
- 20.5 Explique o que os seguintes termos significam na terminologia de banco de dados orientados a objetos: *método*, *assinatura*, *mensagem*, *coleção*, *extensão*.
- 20.6 Qual é o relacionamento entre um tipo e seu subtipo em uma hierarquia de tipos? Qual é a restrição imposta em extensões correspondentes a tipos em hierarquias de tipos?
- 20.7 Qual a diferença entre objetos persistentes e transientes? Como a persistência é manipulada em sistemas de bancos de dados OO típicos?
- 20.8 Como se diferenciam herança simples, herança múltipla e herança seletiva?
- 20.9 Discuta o conceito de polimorfismo/sobrecarga de operador.
- 20.10 Qual a diferença entre objetos complexos estruturados e não estruturados?
- 20.11 Qual a diferença entre semântica de propriedade e semântica de referência em objetos complexos estruturados?
- 20.12 O que é criação de versões? Por que é importante? Qual a diferença entre versões e configurações?

Exercícios

20.13 Converta o exemplo de OBJETOS_GEOMETRICOS apresentado na Seção 20.4 da notação funcional para a notação dada na Figura 20.3, que faz distinção entre atributos e operações. Utilize a palavra-chave INHERIT para mostrar o que uma cia: se herda de outra.

20.14 Compare herança no modelo EER (Capítulo 4) com herança no modelo OO descrito na Seção 20.4-

20.15 Considere o esquema EER UNIVERSIDADE da Figura 4.1. Abstraia quais operações são necessárias para tipos de entidade; des/classes no esquema. Desconsidere as operações construtor e destrutor.

20.16 Considere o esquema ER para EMPRESA da Figura 3.2. Abstraia quais operações são necessárias para tipos de entidade; des/classes no esquema. Desconsidere as operações de construtor e destrutor.

Bibliografia Seleccionada

Os conceitos de bancos de dados orientados a objetos são um amálgama dos conceitos vindos das linguagens de programação OO e dos sistemas de bancos de dados e modelos conceituais de dados. Uma série de livros-textos descreve as linguagens de programação OO — como, por exemplo, Stroustrup (1986) e Pohl (1991) para C++ e Goldberg (1989) para SMALLTALK. Mais recentemente, livros de Cattell (1994), de Lauden e Vossen (1997) descrevem os conceitos de bancos de dados OO. Há vasta bibliografia a respeito de bancos de dados OO, assim, só podemos oferecer aqui uma amostra representativa. A edição de outubro de 1991 da CACM e a edição de dezembro de 1990 da *IEEE Computer* descrevem conceitos e sistemas de bancos de dados orientados a objetos. Dittrich (1986) e Zaniolo et al. (1986) apresentam os conceitos básicos de modelos de dados orientados a objetos. Um dos primeiros artigos sobre bancos de dados orientados a objetos é de Baroudy e DeWitt (1981). Su et al. (1988) apresentam um modelo de dados orientado a objetos utilizando aplicações CAD/CAM. Mitschan (1989) estende a álgebra relacional para abranger objetos complexos. Linguagens de consulta e interfaces gráficas do usuário para OO são descritas em Gyssens et al. (1990), Kim (1989), Alashqur et al. (1989), Bertino et al. (1992), Agrawal et al. (1990) e Cruz (1992).

Polimorfismo em bancos de dados e linguagens de programação orientadas a objetos é discutido em Osborn (1989) Atkinson e Buneman (1987) e Danforth e Tomlinson (1988). Identidade de objetos é discutida em Abiteboul e Kanellak (1989). Linguagens de programação OO para bancos de dados são discutidas em Kent (1991). Restrições de objetos são discutidas em Delcambre et al. (1991) e Elmasri et al. (1993). Segurança e autenticação em bancos de dados OO são examinadas em Rabitti et al. (1991) e Bertino (1992).

Outras referências adicionais são dadas no final do Capítulo 21.

Padrões, Linguagens e Projeto de Banco de Dados de Objetos

Conforme discutimos no início do Capítulo 8, a existência de um padrão para determinado tipo de sistema de banco de dados é muito importante por oferecer suporte à portabilidade de aplicações de bancos de dados. **Portabilidade** é normalmente definida como a capacidade de execução de um programa de aplicação em diferentes sistemas com o mínimo de modificações no programa em si. Na área de bancos de dados de objetos, a portabilidade deveria permitir que um programa escrito para acessar um sistema gerenciador de banco de dados de objetos (SGBDO) acesse um outro SGBDO, de modo que ambos os pacotes suportem fielmente o padrão. Para ilustrar por que a portabilidade é importante, suponha que certo usuário invista milhões de dólares na criação de uma aplicação que é executada em um produto de determinado fornecedor e que ele fique insatisfeito com esse produto por alguma razão — digamos que o desempenho não atenda seus requisitos. Se a aplicação foi escrita com uso de componentes de uma linguagem padrão, é possível que o usuário converta a aplicação para um produto de um outro fornecedor — o qual adere aos mesmos padrões da linguagem, mas que pode possuir melhor desempenho para a aplicação desse usuário — sem ter de fazer grandes modificações que exijam tempo e maiores investimentos financeiros.

Uma segunda vantagem potencial de se possuir padrões e aderir a eles é que estes auxiliam na obtenção da **interoperabilidade**, a qual se refere à habilidade de uma aplicação em acessar diversos sistemas diferentes. Em termos de bancos de dados, isso significa que o mesmo programa de aplicação pode acessar alguns dados armazenados em um SGBDO e outros dados armazenados em um outro pacote. Existem diferentes níveis de interoperabilidade. Por exemplo, os SGBDs poderiam ser dois pacotes diferentes de SGBD do mesmo tipo — dois sistemas de bancos de dados de objetos — ou poderiam ser dois pacotes de SGBDs de tipos distintos — digamos um SGBD relacionai e outro SGBD orientado a objetos. Uma terceira vantagem dos padrões é que eles permitem aos usuários *comparar produtos comerciais*, determinando mais facilmente quais componentes do padrão são oferecidas pelo produto individualmente.

Como foi discutido na introdução do Capítulo 8, uma das razões do sucesso dos SGBDs relacionais comerciais é o padrão SQL. A ausência por vários anos de um padrão para SGBDOs pode ter sido a causa de alguns potenciais usuários terem recuado na conversão para essa nova tecnologia. Subseqüentemente um consórcio de fornecedores de SGBDOs, chamado ODMG (*Object Data Management Group*), apresentou um padrão que é conhecido como ODMG-93 ou ODMG 1.0, o qual foi revisado para o ODMG 2.0, que será descrito neste capítulo. O padrão é composto de vários componentes: o **modelo de objetos**, a **linguagem de definição de objetos (ODL — Object Definition Language)**, a **linguagem de consulta a objetos (OQL — Object Query Language)** e os **acoplamentos (bindings)** para as linguagens de programação orientadas a objetos. Os *bindings* com linguagens têm sido especificados para diversas linguagens orientadas a objetos, incluindo C++, SMALLTALK e JAVA. Alguns fornecedores oferecem *binding* somente para linguagens específicas, sem oferecer todas as capacidades da ODL e da OQL. O modelo

¹ Neste capítulo será utilizado banco de dados de *objeto* em vez de banco de dados *orientado a objetos* (como no capítulo anterior), dado que, atualmente, essa é a terminologia mais usual.

478 Capítulo 21 Padrões, Linguagens e Projeto de Banco de Dados de Objetos

de objetos ODMG será descrito na Seção 21.1, a ODL na Seção 21.2, a OQL na Seção 21.3 e o acoplamento com a linguagem C++ na Seção 21.4. Nos exemplos de como se utilizar a ODL, a OQL e o *binding* com a linguagem C++ será empregado o exemplo do banco de dados UNIVERSIDADE do Capítulo 4. Nessa descrição será seguido o modelo de objetos ODMG 2.0 conforme descrito em Cattel *et al.* (1997). É importante observar que muitas das idéias incorporadas no modelo de objetos do ODMG são baseadas em duas décadas de pesquisa em modelagem conceitual e bancos de dados orientados a objetos, realizadas por diversos pesquisadores. Após a descrição do modelo ODMG, será descrita na Seção 21.5 uma técnica para projeto conceitual de bancos de dados de objetos. Será discutido como os bancos de dados orientados a objetos diferenciam-se dos bancos de dados relacionais e será mostrado como mapear um projeto conceitual de banco de dados no modelo EER para declarações em ODL do modelo ODMG. O leitor pode deixar de ler as seções 21.3 a 21.7 caso deseje uma introdução menos detalhada desses tópicos.

21.1 VISÃO GERAL DO MODELO DE OBJETOS ODMG

O **modelo de objetos** ODMG é um modelo de dados no qual se baseiam a linguagem de definição de objetos (ODL) e a linguagem de consulta a objetos (OQL). De fato, esse modelo de objetos fornece os tipos de dados, os tipos construtores e outros conceitos que podem ser utilizados na ODL para especificar esquemas de bancos de dados de objetos. Assim, ele deve fornecer um modelo de dados padrão para banco de dados orientados a objetos, da mesma forma que o SQL descreve um modelo de dados padrão para bancos de dados relacionais. Ele também fornece uma terminologia padronizada em um contexto no qual às vezes são utilizados os mesmos termos para descrever conceitos distintos. Tentaremos aderir à terminologia ODMG neste capítulo. Muitos dos conceitos do modelo ODMG já foram discutidos no Capítulo 20 e consideramos que o leitor tenha lido as seções 20.1 a 20.5. Faremos indicações toda vez que a terminologia ODMG divergir daquela utilizada no Capítulo 20.

21.1.1 Objetos e Literais

Objetos e literais são os blocos de construção básicos do modelo de objetos. A principal diferença entre os dois é que um objeto possui um identificador do objeto e um **estado** (ou valor atual), enquanto um literal possui somente um valor, mas *não um identificador do objeto*. Em ambos os casos, o valor pode possuir uma estrutura complexa. O estado do objeto pode se alterar ao longo do tempo pela modificação do valor do objeto. Um literal é basicamente um valor constante, podendo possuir uma estrutura complexa, mas que não se modifica.

Um **objeto** é descrito por quatro características: (1) identificador, (2) nome, (3) tempo de vida e (4) estrutura. O identificador do objeto é um identificador único em todo o sistema (ou OBJECT_ID). Todo objeto deve possuir um identificador de objeto. Adicionalmente ao OBJECT_ID, alguns objetos podem opcionalmente receber um **nome** único em certo banco de dados — esse nome pode ser utilizado para referir-se ao objeto em um programa e o sistema deve ser capaz de localizar o objeto por esse nome. Obviamente nem todos os objetos individuais possuirão nomes únicos. Em geral alguns poucos objetos, em especial aqueles que mantêm coleções de objetos de determinado tipo — como extensões — possuirão um nome. Esses nomes são utilizados como **pontos de entrada** no banco de dados, isto é, ao localizar esses objetos pelos seus nomes, o usuário pode localizar outros objetos que são referenciados a partir deles. Outros objetos importantes na aplicação também podem ter nomes únicos. Todos esses nomes dentro de um banco de dados devem ser únicos. O **tempo de vida (lifetime)** de um objeto especifica se ele é um *objeto persistente* (ou seja, um objeto do banco de dados) ou um *objeto transiente* (isto é, um objeto em um programa em execução que desaparece após o término desse programa). Finalmente a **estrutura** de um objeto especifica como ele é construído pelos construtores de tipos. A estrutura define se um objeto é *atômico* ou um *objeto coleção*. O termo *objeto atômico* difere do modo como definimos *construtor atômico* na Seção 20.2.2 e é bastante diferente de um literal atômico (ver a seguir). No modelo ODMG, um objeto atômico é qualquer objeto que não seja uma coleção, desse modo também inclui

2 A primeira versão do modelo de objetos foi publicada em 1993.

3 Serão utilizados aqui os termos 'valor' e 'estado' alternativamente.

4 Corresponde ao OID do Capítulo 20.

5 Isso corresponde ao mecanismo de nomeação descrito na Seção 20.3.2.

6 No modelo ODMG, objetos atômicos não correspondem a objetos cujos valores sejam tipos de dados básicos. Todos os valores básicos (inteiros, reais etc.) são considerados como sendo literais.

21.1 Visão Geral do Modelo de Objetos ODMG 479

objetos estruturados criados com o uso do construtor *struct*. Discutiremos os objetos coleção na Seção 21.1.2 e os objetos atômicos na Seção 21.1.3. Primeiro, será definido o conceito de um literal.

No modelo de objetos, um **literal** é um valor que *não possui* um identificador de objeto. Porém, o valor pode possuir uma estrutura simples ou complexa. Existem três tipos de literais: (1) atômico, (2) coleção e (3) estruturado. Os **literais atômicos** correspondem aos valores de tipos de dados básicos e são predefinidos. Os tipos de dados do modelo de objetos incluem números inteiros longos, curtos e inteiros positivos (que são especificados pelas palavras-chave *Long* (Longo), *Short* (Curto), *Unsigned Long* (Longo não-assinado) e *Unsigned Short* (Curto não-assinado) na ODL), números com ponto flutuante normais e de dupla precisão (*Float*, *Double*), valores lógicos (*Boolean*), caracteres simples (*Char*), cadeias de caracteres (*String*) e tipos enumerados (*Enum*), entre outros. Os **literais estruturados** correspondem basicamente aos valores que são construídos com o uso do construtor de tuplas descrito na Seção 20.2.2. Incluem as estruturas embutidas *Date*, *Interval*, *Time* e *Timestamp* (Figura 21.1b), assim como outras estruturas de tipo adicionais, definidas pelo usuário, necessárias para cada aplicação. As estruturas definidas pelo usuário são criadas utilizando a palavra-chave **Struct** da ODL, assim como nas linguagens de programação C e C++ . Os **literais coleção** especificam um valor que corresponde a uma coleção de objetos ou de valores, mas a coleção propriamente dita não possui um *OBJECT_ID*. AS coleções no modelo de objetos são *SET<T>*, *BAG<T>*, *LIST<T>* e *ARRAY<T>*, em que T é o tipo de objetos ou valores da coleção. Outro tipo de coleção é *Dictionary <K,V>*, o qual corresponde a uma coleção de associações <K,V> na qual cada K é uma chave (um valor único de pesquisa) associada a um valor V; esse tipo pode ser utilizado para criar um índice sobre uma coleção de valores. A Figura 21.1 apresenta uma visão simplificada dos componentes básicos do modelo de objetos. A notação da ODMG usa a palavra-chave *interface* onde havíamos utilizado as palavras-chave *type* (tipo) e *class* (classe) no Capítulo 20. De fato, o termo *interface* é mais apropriado, uma vez que descreve a interface dos tipos de objetos — especificamente seus atributos viáveis, seus relacionamentos e suas operações.

Em geral, essas interfaces são não instanciáveis (isto é, objetos não são criados para uma interface), mas servem para definir operações que podem ser herdadas pelos objetos definidos pelo usuário para determinada aplicação. A palavra-chave *class* no modelo de objetos é reservada para declarações de classes especificadas pelo usuário, que formam um esquema do banco de dados e são utilizadas para criar objetos de aplicação. A Figura 21.1 é uma versão simplificada do modelo de objetos. Para as especificações completas, veja Cattell *et al.* (1997). Vamos descrever os construtores mostrados na Figura 21.1, conforme definidos no modelo de objetos.

```
interface Object {
boolean    same_as(in Object other_object); Object    copy(); void    deletef();
};
```

FIGURA 21.1a Visão geral das definições da interface para parte do modelo de objetos ODMG. A interface básica *Object*, herdada por todos os objetos.

7 O construtor *struct* corresponde ao construtor *tuple* do Capítulo 20.

8 O uso da palavra atômico em *literal atômico* corresponde ao mesmo sentido que utilizamos em *construtor atômico* (*atom*) na Seção 20.2.2.

9 As estruturas para *Date*, *Interval*, *Time* e *Timestamp* podem ser utilizadas para criar outros valores literais ou objetos com identificadores.

10 São similares aos construtores de tipos semelhantes descritos na Seção 20.2.2.

11 Interface também é a palavra-chave utilizada no padrão CORBA (Seção 21.5) e na linguagem de programação JAVA.

480

Capítulo 21 Padrões, Linguagens e Projeto de Banco de Dados de Objetos

```

interface Date : Object {
    enum Weekday
    {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
    enum Month
    {January, February, March, April, May, June, July, August, September, October, November, December};
    unsigned short year();
    unsigned short month();
    unsigned short day();
    boolean boolean
    is_equal(in Date other_Date); is_greater(in Date other_Date);
};
interface Time : Object {
    unsigned short unsigned short unsigned short unsigned short
    hour(); minute(); second(); millisecond();
    boolean boolean
    is_equal(in Time other_Time); is_greater(in Time other_Time);
    Time Time Interval
    add_interval(in Interval some_Interval); subtract_interval(in Interval some_Interval); subtract_time(in Time other_Time);
);
interface Timestamp: Object {
    unsigned short unsigned short unsigned short unsigned short unsigned short unsigned short unsigned short unsigned short
    year();
    month();
    day();
    hour();
    minute();
    second();
    millisecond();
    Timestamp Timestamp boolean boolean
    plus(in Interval someinterval); minus(in Interval someinterval); is_equal(in Timestamp other_Timestamp); is_greater(in Timestamp
    other_Timestamp);
};
interface Interval: Object { unsigned short unsigned short unsigned short unsigned short unsigned short unsigned short
    day();
    hour();
    minute();
    second();
    millisecond();
    Interval Interval Interval Interval boolean boolean
    plus(in Interval someinterval); minus(in Interval someinterval); product(in long some_value); quotient(in long some_value);
    is_equal(in Interval otherinterval); is_greater(in Interval otherinterval);
}

```

FIGURA 21.1 b Visão geral das definições da interface para parte do modelo de objetos ODMC. Algumas in literais estruturados.

21.1 Visão Geral do Modelo de Objetos ODMG

481

```

interface Collection : Object {
    exception ElementNotFound{any element;};
    unsigned long cardinality();
    boolean is_empty();
    boolean contains_element(in any element);
    void insert_element(in any element);
    void remove_element(in any element);
    raises(ElementNotFound); Iterator create_iterator(in boolean stable);
    interface Iterator {
        exception NoMoreElements();
        boolean is__stable();
        boolean at_end();
        void reset();
        any get_element() raises(NoMoreElements);
        void next_position() raises(NoMoreElements)
    }
    interface Set: Collection {
        Set create_union(in Set other_set);
        boolean is_subset_of(in Set other_set);
    };
    interface Bag : Collection {
        unsigned long occurrences_of(in any element);
        Bag create_union(in Bag other_bag);
    };
    interface List: Collection {
        exception InvalidIndex{unsigned_long index;};
        any remove_element,at(in unsigned long position)
    }
}

```

```

raises(InvalidIndex); any
raises(InvalidIndex); void
raises(InvalidIndex); void
raises(InvalidIndex);
void
any
any
List
void
};
insert_elementfirst(in any element);
remove_first_element() raises(InvalidIndex);
retrievefirst_element() raises(InvalidIndex);
concat(in List otherjst);
append(in List otherjst);

```

FIGURA 21.1c Visão geral das definições da interface para parte do modelo de objetos ODMG. Definições de interfaces padrões para objetos coleção. **482** Capítulo 21 Padrões, Linguagens e Projeto de Banco de Dados de Objetos

```

interface Array : Collection {
exception
any
raises(InvalidIndex); any
raises(InvalidIndex); void
raises(InvalidIndex); void
};
Invalid_Index {unsigned long index;};
remove_element_at(in unsigned long index)
retrieve_element_at(in unsigned long index)
replace_element_at(in unsigned long index, in any element)
resize(in unsigned long new_size);
struct Association {any key; any value;};
interface Dictionary : Collection {
exception
void
void
any
boolean
};
KeyNotFoundany key;};
bind(in any key, in any value);
unbind(in any key) raises(KeyNotFound);
lookup(in any key) raises(KeyNotFound);
contains_key(in any key);

```

FIGURA 21.1c Visão geral das definições da interface para parte do modelo de objetos ODMG. Definições de interfaces pad para objetos coleção. (continuação)

No modelo de objetos, todos os objetos herdam a interface básica de Object mostrada na Figura 21.1a. Dessa form; operações básicas que são herdadas por todos os objetos (da interface Object) são: copy (cria uma nova cópia do objeto), dei (exclui o objeto) e same_as (compara a identidade do objeto com outro objeto). Em geral, as operações são aplicadas aos jetos utilizando-se a notação de ponto. Por exemplo, dado um objeto o, para compará-lo a um outro objeto p, escrever

`o.same_as(p)`

O resultado retornado por essa expressão é um valor lógico que pode ser verdadeiro se a identidade de p for a mesma de o e so caso contrário. De modo semelhante, para criar uma cópia p do objeto o escrevemos

`p = o.copy()`

Uma alternativa à notação de ponto é a notação de seta: `o->same_as(p)` ou `o->copy()`.

A herança de tipo que é usada para definir relacionamentos tipo/subtipo é especificada no modelo de objetos com da notação de dois pontos (:), como na linguagem de programação C++ . Assim, na Figura 21.1 pode-se observar que toda interfaces, como Collection, Date e Time, são herdeiras da interface básica Object. No modelo de objetos há dois ti principais de objetos: (1) objetos coleção, descritos na Seção 21.1.2, e (2) objetos atômicos (e estruturados), descrito; Seção 21.1.3.

21.1.2 Interfaces Embutidas para Objetos Coleção

Qualquer objeto coleção é herdeiro da interface básica Collection apresentada na Figura 21.1c, que mostra as operações p todos os objetos coleção. Dado um objeto coleção o, a operação o. cardinality () retorna o número de elementos da cole; A operação o. is_empty () retorna verdadeiro se a coleção for vazia, e falso caso contrário. As operações o. insert_element e o. remove_element (e) incluem ou excluem um elemento e da coleção o. Finalmente, a operação o. contains_element retorna verdadeiro se a coleção o incluir o elemento e, caso contrário, retorna falso. A operação i = o.create_iterator cria um objeto de iteração i para a coleção o, que pode interagir com todos os elementos da coleção. A interface para obj« iterator também é mostrada na Figura 21.1c. A operação i .reset() posiciona o iterativo no primeiro elemento da coleç (para uma coleção não ordenada, este seria um elemento qualquer) e i .next_position() posiciona o iterativo no próxi elemento. A operação i .get_element() recupera o elemento corrente, que é o elemento no qual o iterativo e posicionado naquele momento.

12 Outras operações para locking são definidas para objetos, as quais não são mostradas na Figura 21.1. Discutimos conceitos de locking f bancos de dados no Capítulo 18.

21.1 Visão Geral do Modelo de Objetos ODMG

483

O modelo de objetos ODMG utiliza exceções para relatar erros ou determinadas condições. Por exemplo, a exceção `ElementNotFound` (Elemento não encontrado) na interface `Collection` seria originada pela operação `o.remove_element(e)` caso e não seja um elemento da coleção o. A exceção `NoMoreElements` (Nenhum outro elemento) na interface do iterativo seria gerada pela operação `i.next_position()` caso o iterativo estivesse posicionado naquele momento no último elemento da coleção, não existindo assim nenhum elemento para o iterativo referir-se.

Os objetos coleção são mais especializados como `Set`, `List`, `Bag`, `Array` e `Dictionary`, que herdam as operações da interface `Collection`. Um tipo de objeto `Set<t>` pode ser utilizado para criar objetos de modo que o valor do objeto o seja *um conjunto cujos elementos sejam do tipo t*. A interface para `Set` inclui a operação adicional `p = o.create_union(s)` (Figura 21.1c) a qual retorna um novo objeto p do tipo `Set<t>` que corresponde à união dos dois conjuntos o e s. Outras operações semelhantes à `create_union` (não mostradas na Figura 21.1c) são `create_intersection(s)` e `create_difference(s)`. As operações para comparação de conjuntos incluem a operação `o.is_subset_of(s)`, que retorna verdadeiro se o conjunto de objetos de o for um subconjunto de um outro conjunto de objetos s, e retorna falso caso contrário. Operações semelhantes (não mostradas na Figura 21.1c) são `is_proper_subset_of(s)`, `is_superset_of(s)` e `is_proper_superset_of(s)`. O tipo de objeto `Bag<t>` permite elementos duplicados na coleção e também é herdeiro da interface `Collection`. Ele possui três operações — `create_union(b)`, `create_intersection(b)` e `create_difference(b)` — todas retornando um novo objeto do tipo `Bag<t>`. Por exemplo, `p = o.create_union(b)` retorna um objeto p do tipo `Bag`, que é a união de o e b (armazenando as duplicatas). A operação `o.occurrences_of(e)` retorna o número de ocorrências duplicadas do elemento e na bag o.

Um objeto do tipo `List<t>` herda todas as operações de `Collection` e pode ser utilizado para se criar coleções nas quais a ordem dos elementos seja importante. O valor de cada objeto o é uma lista *ordenada cujos elementos são do tipo t*. Assim, podemos nos referenciar ao primeiro, ao último e ao i-ésimo elemento da lista. Além disso, quando adicionamos um elemento à lista, devemos especificar nela a posição em que o elemento é inserido. Algumas das operações de lista são mostradas na Figura 21.1c. Se o é um objeto do tipo `List<t>`, a operação `o.insert_element_first(e)` (Figura 21.1c) insere o elemento e antes do primeiro elemento na lista o, de modo que e se torna o primeiro elemento na lista. Uma operação semelhante (não mostrada) é `o.insert_element_last(e)`. A operação `o.insert_element_after(e,i)` da Figura 21.1c insere o elemento e após o i-ésimo elemento existente em o e lança a exceção `InvalidIndex` caso não exista o i-ésimo elemento em o. Uma operação semelhante (não mostrada) é `o.insert_element_before(e,i)`. As operações para remover elementos de uma lista são `o.remove_first_element()`, `e = o.remove_last_element()` e `e = o.remove_element_at(i)`; essas operações removem os elementos indicados da lista e retornam o elemento como resultado da operação. Outras operações recuperam um elemento sem removê-lo da lista. São elas `e = o.retrieve_first_element()`, `e = o.retrieve_last_element()` e `e = o.retrieve_element_at(i)`. Por fim, são definidas duas operações para manipular listas. São elas `p = o.concat(l)`, que cria uma nova lista p que é a concatenação das listas o e l (os elementos da lista o seguidos pelos da lista l) e `o.append(i)`, que acresce os elementos da lista i ao final da lista o (sem criar uma nova lista de objetos).

O tipo de objeto `Array<t>` também herda as operações de `Collection`. Um array é semelhante a uma lista, com exceção de que possui um número fixo de elementos. As operações específicas para um objeto `Array` o são `o.replace_element_at(i,e)`, que substitui o elemento do array da posição i pelo elemento e; `e = o.remove_element_at(i)`, que recupera o i-ésimo elemento, trocando-o por um valor nulo; e `e = o.retrieve_element_at(i)`, que simplesmente recupera o i-ésimo elemento do array. Qualquer uma dessas operações pode lançar a exceção `InvalidIndex` caso i seja maior que o tamanho do array. A operação `o.resize(n)` modifica o número de elementos do array para n.

O último tipo de objetos de coleção é o tipo `Dictionary<k,v>`. Esse tipo permite a criação de uma coleção de pares de associação <k,v>, em que os valores de todas as chaves k são únicos. Isso permite a recuperação associativa de determinado par fornecendo-se seu valor de chave (semelhante a um índice). Se o é um objeto coleção do tipo `Dictionary<k,v>`, então `o.bind(k,v)` liga o valor v à chave k como associação <k,v> na coleção, enquanto `o.unbind(k)` exclui a associação com o, e `v = o.lookup(k)` retorna o valor v associado à chave k em o. As duas últimas operações podem lançar exceção `KeyNotFound`. Por fim, `o.contains_key(k)` retorna verdadeiro se a chave k existe em o, e falso caso contrário.

A Figura 21.2 é um diagrama que ilustra a hierarquia de herança das estruturas embutidas do modelo de objetos. As operações são herdadas dos supertipos para os subtipos. As interfaces de objetos coleção descritas acima *não são diretamente instanciáveis*, isto é, não é possível criar objetos baseados nessas interfaces. Por sua vez, as interfaces podem ser utilizadas para especificar objetos coleção definidos pelo usuário — dos tipos `Set`, `Bag`, `List`, `Array` e `Dictionary` — para certa aplicação de banco de dados. Quando um usuário projeta um esquema do banco de dados, irá declarar suas próprias interfaces e classes de objetos que são relevantes à aplicação de banco de dados. Se uma interface ou classe for um dos objetos da coleção, digamos, um `Set`, herdará as operações da interface `Set`. Por exemplo, na aplicação de banco de dados da UNIVERSIDADE, O usuário pode especificar uma classe para `Set<Aluno>`, cujos objetos seriam conjuntos de objetos `Aluno`. O programador pode, então, utilizar as ope-

484

Capítulo 21 Padrões, Linguagens e Projeto de Banco de Dados de Objetos

rações para Set<T> para manipular um objeto do tipo Set<Al uno>. A criação das classes de aplicação normalmente é feita utilizando a linguagem de definição de objetos ODL (Seção 21.2).

Object
Iterator
Interval
Dictionary

FIGURA 21.2 Hierarquia de herança para as interfaces embutidas do modelo de objetos.

É importante observar que todos os objetos de determinada coleção *têm de ser do mesmo tipo*. Assim, embora a palavra-chave *any* (qual quer) apareça nas especificações das interfaces de coleções da Figura 21.1c, isso não significa que objetos de quaisquer tipos possam estar misturados em uma mesma coleção. Ao contrário, isso significa que qualquer tipo pode ser utilizado quando da especificação do tipo de elementos para certa coleção (incluindo outros tipos de coleção!).

21.1.3 Objetos Atômicos (Definidos pelo Usuário)

A seção anterior descreveu os tipos coleções embutidos (*built-in*) do modelo de objetos. Será discutido agora como os tipos objetos para *objetos atômicos* podem ser construídos. Eles são definidos com o uso da palavra-chave **class** na ODL. No modelo de objetos, qualquer um deles definido pelo usuário que não seja um objeto coleção é denominado **objeto atômico**. Por exemplo, no banco de dados da aplicação UNIVERSIDADE, o usuário pode especificar um tipo objeto (classe) para objetos Aluno. A maioria desses objetos será de **objetos** estruturados; por exemplo, um objeto Aluno possuirá uma estrutura complexa, com muitos atributos, relacionamentos e operações, mas ainda será considerado um objeto atômico por não ser uma coleção. Esse tipo de objeto definido pelo usuário é definido como uma classe pela especificação de suas **propriedades e operações**. As propriedades definem o estado do objeto e são, ainda, diferenciadas entre **atributos e relacionamentos**. Nesta subseção, serão tratados os três tipos de componentes — atributos, relacionamentos e operações — que um tipo objeto definido pelo usuário para objetos atômicos (estruturados) pode conter. Essa discussão será ilustrada com as duas classes — Empregado e Departamento — mostradas na Figura 21.3.

Um **atributo** é uma propriedade que descreve algum aspecto de um objeto. Os atributos possuem valores, os quais normalmente são literais que possuem estruturas simples ou complexas armazenadas no objeto. Entretanto, os valores de atributos também podem ser Object_Ids de outros objetos. Os valores de atributos também podem ser especificados por métodos que são utilizados para calcular o valor do atributo. Na Figura 21.3, os atributos para Empregado são nome, ssn, datanascimento, sexo e idade, e para Departamento são dnome, dnumero, ger, local izacoes e projetos. Os atributos ger e projetos de Departamento possuem estrutura complexa e são definidos via **struct**, que corresponde ao *construtor tuple* do Capítulo 20. Assim, o valor de ger em cada objeto Departamento possuirá dois componentes: gerente, cujo valor é um Object_Id que referencia o objeto Empregado que gerencia o Departamento, e datainicio, cujo valor é uma data. O atributo localizações de Departamento é definido pelo construtor set, uma vez que cada objeto Departamento pode possuir um conjunto de localizações.

L3

14

Como mencionado anteriormente, essa definição de *objeto atômico* no modelo de objetos ODMG é diferente da definição do construtor atom apresentada no Capítulo 20, e corresponde à definição adotada na maior parte da literatura sobre bancos de dados orientados a objetos.

Na Figura 21.3 estamos utilizando a notação da Linguagem de Definição de Objetos (ODL), que será discutida mais detalhadamente na Seção 21.2.

21.1 Visão Geral do Modelo de Objetos ODMG

485

```

class Empregado
(
  extent todos_empregados
  key    ssn) {
  attribute    string      nome;
  attribute    string      ssn;
  attribute    date        datanascimento;
  attribute    enum gênero {M,F} sexo;
  attribute    short       Idade;
  relationship Departamento trabalha_em
  inverse Departamento::possuiLempregados;
  void         redesignar_empregado(string novo_dnome)
  raises (dnomejinvalido);
};

class Departamento
(
  extent todos_departamentos
  key    dnome, dnumero) {
  attribute    string      dnome;
  attribute    short       dnumero;
  attribute    struct Ger_Depto {Empregado gerente, date datainicio}
  ger;
  attribute    set<string>   localizações;
  attribute    struct Projetos {string nomeprojeto, time horas_semana}
  projetos;
  relationship set<Empregado> possui_empregados inverse Empregado::trabalha_em;
  void         incluir_empregado(in string novo_ename) raises(enamejinvalido);
  void         alterar_gerente(in string novo_nome_ger; in date datainicio);
};

```

FIGURA 21.3 Atributos, relacionamentos e operações em definição de classe.

Um relacionamento é uma propriedade que especifica que dois objetos em um banco de dados estão conjuntamente relacionados. No modelo de objetos ODMG, somente relacionamentos binários (Capítulo 3) são representados explicitamente, e cada relacionamento binário é representado por um *par de referências inversas* definidas pela palavra-chave *relationship*. Na Figura 21.3 existe um relacionamento que associa cada Empregado ao Departamento no qual ele trabalha — o relacionamento *trabalha_em* de Empregado. Na direção inversa, cada Departamento é relacionado ao conjunto de Empregados que trabalham no Departamento — o relacionamento *possui_empregados* de Departamento. A palavra-chave *inverse* especifica que essas duas propriedades definem um único relacionamento conceitual em direções inversas. Por essa especificação, o sistema de banco de dados pode manter a integridade referencial do relacionamento automaticamente. Isto é, se o valor de *trabalha_em* para determinado Empregado e se refere ao Departamento *d*, então o valor de *possui_empregados* para o Departamento *d* deve incluir uma referência para e em seu conjunto de referências para Empregados. Se o projetista do banco de dados desejar definir um relacionamento em *uma única direção*, o mesmo deve ser modelado como um atributo (ou operação). Um exemplo é o componente gerente do atributo *ger* em Departamento.

Além de atributos e relacionamentos, o projetista pode incluir operações em especificações de tipos (ou classes) de objetos. Cada tipo objeto pode possuir um número de assinaturas de operação, as quais especificam o nome da operação, seus tipos de argumento e seu valor de resposta, se aplicável. Os nomes das operações são únicos para cada tipo objeto, mas eles podem ser repetidos se existir o mesmo nome de operação aparecendo em diferentes tipos objetos. A assinatura da operação também pode especificar nomes de exceções que podem ocorrer durante a execução do programa. A implementação da operação incluirá o código referente a essas exceções. Na Figura 21.3, a classe Empregado possui uma operação, *redesignar_empregado*, e a classe Departamento, duas operações, *incluir_empregado* e *alterar_gerente*.

21.1.4 Interfaces, Classes e Herança

No modelo de objetos ODMG, existem dois conceitos para especificar tipos objetos: interfaces e classes. Além disso, há dois tipos de relacionamentos de herança. Nesta seção serão discutidas as diferenças e similaridades entre esses conceitos. Seguindo

15 O Capítulo 3 abordou como um relacionamento pode ser representado por dois atributos em direções inversas.

a terminologia ODMG, será utilizada a palavra **comportamento** para referenciar *operações*, e **estado** para referenciar *propriedades* (atributos e relacionamentos).

Uma **interface** é uma especificação do comportamento abstrato de um tipo objeto, que especifica as assinaturas das operações. Embora uma interface possa possuir propriedades de estado (atributos e relacionamentos) como parte de sua especificação, estas *não podem* ser herdadas da interface, conforme veremos. Uma interface também é **não instanciável** — isto é, não podem ser criados objetos que correspondam à definição da interface.

Uma **classe** é a especificação tanto do comportamento abstrato como do estado abstrato de um tipo objeto e é **instanciável** — ou seja, pode criar instâncias de objetos individuais correspondentes à definição da classe. Por não serem instanciáveis, as interfaces são utilizadas principalmente para especificar operações abstratas que podem ser herdadas por

17 classes ou por outras interfaces. Isso é chamado **herança de comportamento** e é especificado pelo símbolo ":". Assim, no modelo de objetos ODMG a herança de comportamento exige que o supertipo seja uma interface, enquanto o subtipo pode ser uma classe ou uma outra interface.

Outro relacionamento de herança, chamado EXTENSÃO e especificado pela palavra-chave **extends**, é utilizado para herdar estado e comportamento estritamente entre classes. Em uma herança de EXTENSÃO, tanto o supertipo como o subtipo devem ser classes. A herança múltipla por EXTENSÃO não é permitida. Porém, a herança múltipla é permitida para herança de comportamento pelo símbolo ":". Assim, uma interface pode herdar comportamentos de várias outras interfaces. Uma classe pode também herdar comportamento de outras interfaces via ":", além de herdar comportamento e estado de, *no máximo, uma* outra classe por EXTENSÃO. Na Seção 21.2 serão apresentados exemplos de como esses dois relacionamentos de herança — ":" e EXTENSÕES — podem ser utilizados.

21.1.5 Extensões, Chaves e Objetos Fábrica

No modelo de objeto ODMG, o projetista de banco de dados pode declarar uma **extensão (extent)** para qualquer tipo de objeto que esteja definido por meio de uma declaração de **classe (class)**. Um nome é atribuído à extensão, e esta irá conter todos os objetos persistentes dessa classe. Assim, uma extensão comporta-se como um conjunto de objetos que mantém todos os objetos persistentes da classe. Na Figura 21.3, as classes Empregado e Departamento possuem extensões chamadas todos_empregados e todos_departamentos, respectivamente. Isso é semelhante à criação de dois objetos — o primeiro do tipo Set<Empregado> e o segundo do tipo Set<Departamento> — e torna-os persistentes nomeando-os como todos_empregados e todos_departamentos. As extensões também são utilizadas para impor automaticamente o relacionamento conjunto/subconjunto entre as extensões de um supertipo e seu subtipo. Se duas classes A e B têm como extensões todos_A e todos_B, e a classe B é um subtipo da classe A (isto é, a classe B EXTENDS a classe A), então, em qualquer momento, a coleção de objetos em todos_B deve ser um subconjunto dos elementos em todos_A. Essa condição é automaticamente imposta pelo sistema de banco de dados.

Uma classe com extensão pode possuir uma ou mais chaves. Uma **chave** consiste em uma ou mais propriedades (atributos ou relacionamentos) cujos valores estão restritos a serem únicos para cada objeto na extensão. Por exemplo, na Figura 21.3, a classe Empregado possui o atributo ssn como chave (cada objeto Empregado na extensão deve possuir um único valor para ssn) e a classe Departamento possui duas chaves distintas: dnome e dnumero (cada Departamento deve possuir um único dnome e um único dnumero). Para uma chave composta que é formada por várias propriedades, as propriedades que compõem a chave estão contidas entre parênteses. Por exemplo, se uma classe Veículo com uma extensão todos_veículos possui uma chave formada pela combinação dos atributos estado e numero_1 i cenca, estes devem estar colocados entre parênteses na forma (estado, numero_1 i cenca) na declaração da chave.

A seguir, apresentamos o conceito de **objeto fábrica** — um objeto que pode ser utilizado para gerar ou criar objetos individuais por meio de suas operações. Algumas das interfaces dos objetos fábrica que fazem parte do modelo de objetos ODMG são mostradas na Figura 21.4. A interface ObjectFactory possui uma única operação, new(), que retorna um novo objeto com um Object_Id. Por meio da herança dessa interface, os usuários podem criar suas próprias interfaces fábrica para cada tipo de objeto definido pelo usuário (atômico), e o programador pode implementar a operação new diferentemente para cada tipo de objeto. A Figura 21.4 também mostra uma interface DateFactory (Fabrica deDatas), que possui operações adicionais para criar um novo calendar_date (data_calendario) e um objeto cujo valor corresponde à data atual (current_date), entre outras

16 Essa definição é similar ao conceito de classe abstrata na linguagem de programação C++.

17 O relatório do ODMG chama também herança de interface como relacionamentos tipo/subtipo, é-um e generalização/especialização, apesar de a literatura utilizar esses termos para descrever herança tanto de estado como de operações (capítulos 4 e 20).

18 Uma chave composta é chamada *chave combinada* no relatório do ODMG.

21.2 ODL — A Linguagem de Definição de Objetos 487

operações (não mostradas na Figura 21.4). Como podemos observar, um objeto fábrica basicamente oferece suporte a operações de construtores para novos objetos.

Finalmente, discutimos o conceito de banco de dados. Pelo fato de um SGBDO poder criar muitos bancos de dados diferentes, cada um com seu próprio esquema, o modelo de objetos ODMG possui as interfaces para objetos `DatabaseFactory` e `Database`, conforme mostrado na Figura 21.4. Cada banco de dados possui seu próprio identificador de banco de dados e a operação **bind** pode ser utilizada para atribuir nomes individuais únicos para objetos persistentes em determinado banco de dados. A operação **lookup** retorna um objeto de um banco de dados que possui um nome de objeto (`object_name`) específico e a operação **unbind** exclui o nome de um objeto persistente especificado no banco de dados.

```
interface ObjectFactory { Object      new();
};
interface DateFactory : ObjectFactory { exception    InvalidDate{};
Date        calendar_date( in unsigned short year,
in unsigned short month, in unsigned short day) raises(InvalidDate);
Date        current();
};
interface DatabaseFactory { Database      new();
};
interface Database {
void        open(in string database_name);
void        close();
void        bind(in any some_object, in string object_name);
Object      unbind(in string name);
Object      lookup(in string object_name)
raises(ElementNotFound);
};
```

FIGURA 21.4 Interfaces para ilustrar objetos fábrica e objetos de banco de dados (*database*).

21.2 ODL — A LINGUAGEM DE DEFINIÇÃO DE OBJETOS

Após a visão geral, na seção anterior, do modelo de objetos ODMG, mostraremos agora como esses conceitos podem ser empregados para criar um esquema de banco de dados de objetos utilizando a linguagem de definição de objetos ODL. A ODL é projetada para dar suporte aos construtores semânticos do modelo de objetos ODMG e é independente de qualquer linguagem de programação em particular. Seu principal uso consiste em criar especificações de objetos — isto é, classes e interfaces. Assim, a ODL não é uma linguagem de programação completa. Um usuário pode especificar um esquema de banco de dados em ODL independentemente de qualquer linguagem de programação, para então utilizar o *binding* específico com a linguagem para especificar como os componentes em ODL podem ser mapeados para componentes em linguagens de programação específicas, como C++ , SMALLTALK e JAVA. Uma visão geral do *binding* com a linguagem C++ será mostrada na Seção 21.4.

A Figura 21.5b mostra um possível esquema de objetos para parte do banco de dados UNIVERSIDADE, o qual foi apresentado no Capítulo 4. Os conceitos da ODL serão descritos utilizando-se este exemplo e outro da Figura 21.7. A notação gráfica para a Figura 21.5b é mostrada na Figura 21.5a e pode ser considerada uma variação dos diagramas EER (Capítulo 4) com a adição do conceito de herança de interface, mas sem vários conceitos do EER, como categorias (tipos *union*) e atributos de relacionamentos.

A Figura 21.6 apresenta um possível conjunto de definições de classes em ODL para o banco de dados UNIVERSIDADE. Geralmente pode haver vários possíveis mapeamentos de um diagrama de esquema de objetos (ou um diagrama de esquema EER) para classes ODL. Discutiremos essas opções mais adiante, na Seção 21.5.

19 A sintaxe e os tipos de dados da ODL devem ser compatíveis com a linguagem de definição de interface (IDL) do CORBA (Common *Object Request Broker Architecture*) com extensões para relacionamentos e outros conceitos de bancos de dados.

A Figura 21.6 mostra um modo direto para se mapear parte do banco de dados UNIVERSIDADE do Capítulo 4. Os tipos de entidades são mapeados para classes ODL, e a herança é definida com o uso de *EXTENDS*. Entretanto, não há nenhum método para mapear categorias (tipos *union*) ou para tratar herança múltipla. Na Figura 21.6, as classes Pessoa, Professor, Al uno e GradAl uno possuem as extensões Pessoas, Professores, Al unos e GradAl unos, respectivamente. Professor e Al uno são herdeiros (EXTENDS) de Pessoa, enquanto GradAl uno herda (EXTENDS) de Al uno. Desse modo, a coleção de Al uno (e a coleção de Professores) estará restrita a ser um subconjunto da coleção de Pessoa em dado momento. Do mesmo modo, a coleção de GradAl uno será um subconjunto de Al unos. Ao mesmo tempo, objetos individuais de Al uno e Professores herdarão as propriedades (atributos e relacionamentos) e operações de Pessoa, e os objetos individuais de GradAl uno serão herdeiros dos objetos da classe Al uno equivalentes.

As classes Departamento, Curso, Disciplina e DisciplinaOferecida na Figura 21.6 são diretamente mapeadas a partir dos tipos de entidade correspondentes da Figura 21.5b. Porém, a classe Nota exige alguma explicação. A classe Nota corresponde ao relacionamento M:N entre Al uno e Disciplina na Figura 21.5b. O motivo de se definir uma classe separada (em vez de um par de relacionamentos inversos) é que ela inclui o atributo de relacionamento Nota.

Interface
Class
Aluno
Relacionamentos



1:1 1:N M:N
Herança \ Herança de
(a)
interface utilizando":

i

Herança de
classe
utilizando **extends**
Pessoa

-----£---, Estudante
Professor v
tem_especializacao
possuiProfessores
Departamento

>' especializa_em 1/ disciplinas_concluidas --<-----
n i \ orienta

orientado. ---*~+-

-P_or X_o
matriculados_em
na_banca_de
banca
Grad Aluno
oferece
oferecido^por i r
Curso
possui_disciplinas a
>' alunos
Disciplina
(b)
alunos_matriculados

I

do_curso
Disciplina_oferecida

FIGURA 21.5 Um exemplo de um esquema de banco de dados, (a) Notação gráfica para representação de esquemas ODL. (b) Esquema gráfico de banco de dados de objetos para parte do banco de dados UNIVERSIDADE.

20 Discutiremos mapeamentos alternativos para atributos de relacionamento na Seção 21.5.

21.2 ODL — A Linguagem de Definição de Objetos

489

```

class Pessoa (   extent pessoas key   ssn)
{
  attribute struct Pnome {string pnome, string mnome, string unome }
  nome; attribute string          ssn;
  attribute date                 datanascimento;
  attribute enum Qgenero{M, F}   sexo; attribute struct Endereço
  {string rua, short nro, short nroapto, string cidade, string estado, short CEP }
  endereço; short idade();
};
class Professor extends Pessoa (   extent professores)
{
  attribute string      classificação;
  attribute float       salário;
  attribute string      sala;
  attribute string      telefone;
  relationship Departamento trabalha_em inverse Departamento::possuLprofessor;
  relationship set<Grad_Aluno > orienta inverse Grad_Aluno::orientado_por;
  relationship set<Grad_Aluno > na_banca_de
  inverse Grad_Aluno::banca; void dar_aumento(in float aumento); void promover(in string nova_classificacao);
};
class Nota
(   extent notas)
{
  attribute enum Valor_nota{A,B,C,D,F,I,P} nota;
  relationship Disciplina disciplina inverse Disciplina::alunos;
  relationship Aluno aluno inverse Aluno::disciplinas_concluidas;
};
class Aluno extends Pessoa
(   extent alunos )
{
  attribute string      categoria;
  attribute Departamento estuda_em;
  relationship Departamento especializa_em inverse Departamento::forma_especialistas;
  relationship set<Nota> disciplinas_completadas inverse Nota::aluno;
  relationship set<Disciplina_oferecida> matriculado_em
  inverse Disciplina_oferecida::alunos_matriculados;
  void alterar_especializacao (in string dnome) raises (nome_departamento_invalido);
  float mdc(); /*média de disciplinas concluídas */
  void matricular(in short cod_disciplina) raises (disciplina_invalida);
  void atribui_avaliacao(in short cod_disciplina; in Valor_Nota nota)
  raises (disciplina_invalida, nota_invalida);
  class Grau
  {
    attribute string      faculdade;
    attribute string      grau;
    attribute string      ano;
  };
  class Grad_Aluno extends Aluno

```

FIGURA 21.6 Um possível esquema ODL para o banco de dados UNIVERSIDADE da Figura 21.5b.

490 Capítulo 21 Padrões, Linguagens e Projeto de Banco de Dados de Objetos

```
( extent Grad^Aluno ) {
  attribute set<Grau> graus;
  relationship Professor orientado_por inverse Professor: :orienta; relationship set<Professor> banca inverse Professor::na_banca_de; void
  atribuiOrientador(in string unome; in string pnome)
  raises (professorjnválido); void nomeia_membro_banca(in string unome; in string pnome) raises (professorjnválido);
};
class Departamento
( extent departamentos key dnome )
(
  attribute string dnome;
  attribute string dtelefone;
  attribute string dsala;
  attribute string dfaculdade;
  attribute Professor coordenador;
  relationship set<Professor> possui_professores inverse Professor: :trabalha_em;
  relationship set <Aluno> forma_especialistas inverse Aluno::especializa_em;
  relationship set<Curso> oferece inverse Curso::oferecido_em;
);
class Curso
( extent cursos key nroc )
{
  attribute string nomec;
  attribute string nroc;
  attribute string descricao;
  relationship set<Disciplina> possui_disciplinas inverse Disciplina::do_curso;
  relationship Departamento oferecido_por inverse Departamento: :oferece;
};
class Disciplina
( extent disciplinas )
{
  attribute short nrod;
  attribute string ano;
  attribute enum Trimestre(Primeiro, Segundo, Terceiro, Quarto) trim;
  relationship set<Nota> alunos inverse Nota::disciplina;
  relationship Curso do_curso inverse Curso::possui_disciplinas;
};
class Disciplina_oferecida extends Disciplina
( extent disciplinas,,oferecidas)
{
  relationship set<Aluno> alunos_matriculados inverse Aluno::matriculado_em;
  void matricular_aluno(in string ssn)
  raises(aluno_invalido, disciplina_sem_vagas);
};
```

FIGURA 21.6 Um possível esquema ODL para o banco de dados, UNIVERSIDADE da Figura 21.5b (continuação).

Dessa forma, o relacionamento M:N é mapeado para a classe Nota, e um par de relacionamentos 1:N, entre Aluno e Nota e outro entre Disciplina e Nota. Esses dois relacionamentos são representados pelas seguintes propriedades de relacionamento: disciplinas_concluidas de Aluno, disciplina e aluno de Nota e alunos de Disciplina (Figura 21.6). Finalmente, a classe Grau é utilizada para representar o atributo composto multivalorado Grau de Aluno (Figura 4.10).

21 Semelhante ao modo como um relacionamento M:N é mapeado no modelo relacional (Capítulo 7) e no modelo de rede (Apêndice C).

21.2 ODL — A Linguagem de Definição de Objetos

491

Em razão de o exemplo anterior não incluir nenhuma interface, mas somente classes, apresentaremos agora um outro exemplo para ilustrar interfaces e herança de interface (comportamento). A Figura 21.7 é parte de um esquema de banco de dados para armazenar objetos geométricos. Uma interface ObjetoGeometrico é especificada, com operações para calcular o perímetro e a área de um objeto geométrico, mais as operações para transladar (mover) e rotacionar um objeto. Outras classes (Retangulo, Triangulo, Circulo, ...) são herdeiras da interface ObjetoGeometrico. Dado que ObjetoGeometrico é uma interface, ela é não instanciável — ou seja, nenhum objeto pode ser criado baseado diretamente nessa interface. Contudo, objetos do tipo Retangulo, Triangulo, Circulo ... podem ser criados e esses objetos herdam todas as operações da interface ObjetoGeometrico. Observe que, com a herança de interface, somente operações são herdadas, não propriedades (atributos ou relacionamentos). Desse modo, se uma propriedade é necessária em uma herança de classes, deve ser repetida na definição de classes, como o atributo ponto_de_referencia da Figura 21.7. Deve ser notado que a herança de operações pode possuir diferentes implementações em cada classe. Por exemplo, as implementações de área e perímetro podem ser diferentes para Retangulo, Triangulo e Circulo.

ObjetoGeometrico

Retangulo

Triangulo

Circulo

FIGURA 21.7a Uma ilustração de herança de interface por meio de ":". Representação gráfica do esquema.

E permitida a *herança múltipla* de interfaces para uma classe, assim como herança múltipla de interfaces por outra interface. Entretanto, com herança com EXTENDS (herança de classe), a herança múltipla *não é permitida*. Assim, uma classe pode ser herdeira, por meio de EXTENDS, de no máximo uma classe (além de herdar de nenhuma até muitas interfaces).

interface ObjetoGeometrico I

attribute **enum** Formato{Retangulo,Triangulo,Circulo...} formato;

attribute **struct** Ponto {**short** x, **short** y} ponto_de_referencia;

float perimetroO;

float area();

void transladar(in **short** x_translacao; in **short** y_translacao);

void rotacionar(in **float** angulo_de_rotacao);

};

class Retangulo : ObjetoGeometrico

(extent retangulos)

{

attribute **struct** Ponto {**short** x, **short** y} ponto_de_referencia;

attribute **short** comprimento;

attribute **short** altura;

attribute **float** angulo_de_orientacao;

};

class Triangulo : ObjetoGeometrico

(extent triângulos)

{

attribute **struct** Ponto {**short** x, **short** y} ponto_de_referencia;

attribute **short** lado_1;

attribute **short** lado_2;

attribute **float** angulo_lado1_lado2;

attribute **float** angulo_de_orientacao_lado1;

FIGURA 21.7b Ilustração de herança de interface utilizando ":". Definições de interfaces e classes correspondentes em ODL.

```

};
class Circulo : ObjetoGeometrico ( extent círculos )
{
attribute struct Ponto {short x, short y} ponto_de_referencia; attribute short raio;
};

```

FIGURA 21.7b Ilustração de herança de interface utilizando ":". Definições de interfaces e classes correspondentes em ODL (cont.).

21.3 A LINGUAGEM DE CONSULTA DE OBJETOS (OQL)

A linguagem de consulta de objetos (OQL) é proposta para o modelo de objetos ODMG. É projetada para trabalhar acoplada com as linguagens de programação com as quais o modelo ODMG define um *binding*, como C++, SMALLTALK e JAVA. Assim, uma consulta OQL embutida em uma dessas linguagens de programação pode retornar objetos compatíveis com o sistema de tipos dessa linguagem. Além disso, as implementações das operações de classe em um esquema ODMG podem ter seus códigos escritos nessas linguagens de programação. A sintaxe OQL para consultas é semelhante à sintaxe da linguagem padrão de consulta relacional SQL, com características adicionais para conceitos ODMG, como identidade de objetos, objetos complexos, operações, herança, polimorfismo e relacionamentos.

Vamos primeiro discutir a sintaxe de consultas OQL simples e o conceito de utilização de objetos ou extensões com nomes como pontos de entrada no banco de dados, vistos na Seção 21.3.1. Em seguida, na Seção 21.3.2, discutiremos a estrutura de resultados de consultas e o uso de expressões de caminho para percorrer relacionamentos entre objetos. Outros recursos da OQL para manipular identidade de objetos, herança, polimorfismo e outros conceitos orientados a objetos são discutidos na Seção 21.3.3. Os exemplos para ilustrar consultas OQL baseiam-se no esquema do banco de dados UNIVERSIDADE apresentado na Figura 21.6.

21.3.1 Consultas OQL Simples, Pontos de Entrada do Banco de Dados e Variáveis de Iteração

A sintaxe básica da OQL é uma estrutura **sei ect... f rom... where...**, igual a da SQL. Por exemplo, a consulta para recuperar os nomes de todos os departamentos na faculdade de 'Engenharia' pode ser escrita como:

```

QO: SELECT d.dnome
FROM d in departamentos
WHERE d.faculdade = 'Engenharia';

```

Em geral, é necessário um **ponto de entrada** para o banco de dados para cada consulta, que pode ser qualquer *objeto persistente nomeado*. Para muitas consultas, o ponto de entrada é o nome da **extensão** de uma classe. Recorde-se de que o nome da extensão é considerado como sendo o nome de um objeto persistente cujo tipo seja uma coleção (na maioria dos casos, um conjunto) de objetos da classe. Observando os nomes de extensões na Figura 21.6, o objeto nomeado departamentos é do tipo set<Departamento>; pessoas é do tipo set<Pessoa>; professores é do tipo set<Professor>; e assim por diante.

A utilização de um nome de extensão — departamentos em QO — como um ponto de entrada refere-se a uma coleção persistente de objetos. Sempre que uma coleção for referenciada em uma consulta OQL, devemos definir uma **variável de iteração** — d em QO — que percorre cada objeto na coleção. Em muitos casos, como em QO, a consulta irá selecionar certos objetos da coleção, com base nas condições especificadas na cláusula where. Em QO, somente objetos persistentes d na coleção de departamentos que satisfazem a condição d.faculdade = 'Engenharia' são selecionados para o resultado da consulta. Para cada objeto selecionado d, o valor de d.dnome é recuperado no resultado da consulta. Assim, o *tipo do resultado* para QO é bag<string> porque o tipo de cada valor dnome é uma string (embora o verdadeiro resultado seja um set porque dnome é um atributo-chave). Em geral, o resultado de uma consulta seria do tipo twgpara sei ect... f rom... e do tipo set para sei ect distinct... from..., como na SQL (a adição da palavra-chave distinct elimina duplicatas).

Utilizando-se o exemplo em QO, existem três opções sintáticas para especificar variáveis de iteração:

```

d in departamentos

```

22 É similar às variáveis de tupla que percorrem as tuplas nas buscas da SQL.

21.3 A Linguagem de Consulta de Objetos (OQL) 493

departamentos d
departamentos as d

Utilizaremos a primeira construção em nossos exemplos.

Os objetos nomeados utilizados como pontos de entrada para consultas em OQL não se limitam a nomes de extensões. Qualquer objeto persistente nomeado que se refira a um objeto atômico (simples) ou a um objeto coleção pode ser utilizado como um ponto de entrada para o banco de dados.

21.3.2 Resultados de Consultas e Expressões de Caminho

Em geral, o resultado de uma consulta pode ser de qualquer tipo que possa ser expresso no modelo de objetos ODMG. Uma consulta não precisa seguir necessariamente a estrutura `select... from... where...`; no caso mais simples, qualquer nome persistente por si mesmo é uma consulta, cujo resultado é uma referência a esse objeto persistente. Por exemplo, a consulta:

Q1: departamentos;

retorna uma referência à coleção de todos os objetos persistentes de departamento, cujo tipo seja `set<Departamento>`.

Analogamente, suponha que tivéssemos atribuído (pela operação de ligação com o banco de dados — Figura 21.4) um nome persistente `departamentocc` a um único objeto de departamento (o departamento de ciência da computação); então, a consulta:

Q1a: `departamentocc`;

retorna uma referência a um objeto individual do tipo `Departamento`. Uma vez que um ponto de entrada é especificado, o conceito de **expressão de caminho** (*path expression*) pode ser utilizado para indicar o caminho de atributos e objetos relacionados.

Normalmente, uma expressão de caminho inicia-se pelo *nome de um objeto persistente* ou pela variável de iteração que percorre os objetos individuais de uma coleção. Esse nome será acompanhado por zero ou mais nomes de relacionamentos ou nomes de atributos conectados utilizando-se a *notação de ponto*. Por exemplo, referindo ao banco de dados UNIVERSIDADE da Figura 21.6, mostramos, a seguir, exemplos de expressões de caminho, que também são consultas válidas em OQL:

Q2 : `departamentocc.coordenador`;

Q2a: `departamentocc.coordenador.classificacao`;

Q2b: `departamentocc.possui_professores`;

A primeira expressão Q2 retorna um objeto do tipo `Professor`, pois é o tipo do atributo `coordenador` da classe `Departamento`. Esta será uma referência ao objeto `Professor` que está relacionado ao objeto `Departamento` cujo nome persistente é `departamentocc` pelo atributo `coordenador`, ou seja, uma referência ao objeto `Professor` que é a pessoa que coordena o departamento de ciências da computação. A segunda expressão Q2a é semelhante, exceto que retorna a classificação do objeto `Professor` (o coordenador de ciências da computação) em vez da referência ao objeto; assim, o tipo retornado por Q2a é uma string, que é o tipo de dado do atributo `cl` assim fi cacaio da classe `Professor`.

As expressões de caminho Q2 e Q2a retornam valores simples, pois os atributos `coordenador` (de `Departamento`) e `classificacao` (de `Professor`) são ambos mono valorados e aplicáveis a objetos simples. A terceira expressão, Q2b, é diferente; ela retorna um objeto do tipo `set<Professor>` mesmo quando aplicada a um objeto simples, pois é o tipo do relacionamento `possui_professor` da classe `Departamento`. A coleção retornada incluirá as referências a todos os objetos `Professor` que estão relacionados ao objeto persistente cujo nome persistente for `departamentocc` por meio do relacionamento `possui_professor`; ou seja, as referências a todos os objetos `Professor` que trabalham no departamento de ciências da computação. Porém, para retornar as classificações dos professores de ciências da computação *não podemos* escrever

Q3': `departamentocc.possui_professores.classificacao`;

Isso ocorre porque não está claro se o objeto retornado seria do tipo `set<string>` ou `bag<string>` (sendo a última mais provável, uma vez que vários professores podem possuir a mesma classificação). Em razão desse tipo de problema de ambigüidade, a OQL não permite expressões como Q3'. Para tal, deve ser utilizada uma variável de iteração sobre essas coleções como em Q3a e Q3b a seguir:

23 Observe que as duas últimas opções são semelhantes à sintaxe para especificar variáveis de tuplas em consultas SQL.

Q3a: **select** f.classificação

from f **in** departamentocc.possui_professores;

Q3b: **select distinct** f.classificação

from f **in** departamentocc.possui_professores;

Nesse caso, Q3a retorna uma *bag*<string> (valores de classificação duplicados aparecem no resultado) enquanto Q3b retorna um *set*<string> (os valores duplicados são excluídos por meio da palavra-chave *distinct*). Ambas, Q3a e Q3b, ilustram como uma variável de iteração pode ser definida na cláusula *from* para abranger determinada coleção especificada na consulta. A variável *f* em Q3a e Q3b abrange os elementos da coleção *departamentocc.possui_professores*, que é do tipo *set*<Professor>, e inclui apenas aqueles professores que são membros do departamento de ciência da computação.

Em geral, uma consulta OQL pode retornar um resultado com uma estrutura complexa especificada na própria consulta, pela utilização da palavra-chave *struct*. Considere os dois exemplos que seguem:

Q4: *departamentocc.coordenador.orienta*;

Q4a: **select struct** (nome: *struct*(ultimo_nome: s.nome.unome,

primeiro_nome: s.nome.pnome), graus:(**select struct** (grau: d.grau, ano: d.ano,

faculdade: d.faculdade) **from** d **in** s.graus) **from** s **in** departamentocc.coordenador.orienta;

Aqui, Q4 é direto, retornando um objeto do tipo *set*<Grad_Aluno> como seu resultado; essa é a coleção dos alunos de pós-graduação que são orientados pelo coordenador do departamento de ciências da computação. Agora, suponha que seja necessária uma consulta para recuperar os primeiros e os últimos nomes desses alunos de pós-graduação, mais a lista das graduações anteriores de cada um. Isso pode ser escrito como em Q4a, em que a variável *s* corresponde à coleção dos alunos de pós-graduação orientados pelo coordenador, e a variável *d* corresponde aos graus de cada aluno *s*. O tipo do resultado de Q4a é uma coleção de *structs* (em primeiro nível) em que cada *struct* possui dois componentes: *nome* e *graus*. O componente *nome* é uma outra estrutura formada pelo último e pelo primeiro nome, cada um sendo uma cadeia de caracteres simples. O componente *graus* é definido pela consulta embutida, sendo ele próprio uma coleção de *structs* adicionais (em segundo nível), cada uma com três strings como componentes: *grau*, *ano* e *escola*.

Observe que a OQL é *ortogonal* no que se refere à especificação de expressões de caminho. Isto é, os atributos, os relacionamentos e os nomes de operações (métodos) podem ser utilizados alternativamente dentro de expressões de caminho, enquanto não for comprometido o sistema de tipos da OQL. Por exemplo, podem-se escrever as seguintes consultas para recuperar a média das disciplinas cursadas (*mdc*) de todos os alunos da categoria 'superior' formando-se em ciências da computação, com o resultado ordenado por *mdc* constando o primeiro e o último nome:

Q5a: **selectstruct** (ultimo_nome: s.nome.unome,

primeiro_nome: s.nome.pnome, mdc: s.mdc) **from** s **in** departamentocc.forma_especialistas **where** s.categoria = 'superior'

order by mdc **desc**, ultimo_nome **asc**, primeiro_nome **asc**; Q5b: **select struct** (ultimo_nome: s.nome.unome,

primeiro_nome: s.nome.pnome, mdc: s.mdc) **from** s **in** alunos **where** s.especializa_em.dnome = 'Ciências da Computação'

and

s.categoria = 'superior' **order by** mdc **desc**, ultimo_nome **asc**, primeiro_nome **asc**;

Q5a utiliza *departamentocc* como ponto de entrada nomeado para localizar diretamente a referência ao departamento de ciências da computação e depois localizar os alunos por meio do relacionamento *forma_especialistas*, enquanto Q5b pesquisa a extensão *alunos* para localizar todos os estudantes que se especializam naquele departamento. Observe como os nomes de atributos, os relacionamentos e as operações (métodos) são utilizados alternativamente (de modo ortogonal) em expressões de caminho: *mdc* é uma operação; *especializa_em* e *forma_especialistas* são relacionamentos; e *categoria*, *nome*, *dnome*, *pnome* e *unome* são atributos. A implementação da operação *mdc* calcula a média de disciplinas cursadas e retorna esse valor como um tipo ponto flutuante para cada aluno selecionado.

24 Como mencionado anteriormente, *struct* corresponde ao construtor de tuplas apresentado no Capítulo 20.

21.3 A Linguagem de Consulta de Objetos (OQL)

495

A cláusula *order by* é semelhante à estrutura correspondente em SQL e especifica em qual ordem o resultado da consulta deve ser apresentado. Por conseguinte, a coleção que é retornada por uma consulta que contém uma cláusula *order by* é do tipo *list*.

21.3.3 Outras Características da OQL

Especificando Visões como Consultas Nomeadas. O mecanismo de visão na OQL utiliza o conceito de consulta nomeada. A palavra-chave *define* é utilizada para especificar um identificador para a consulta nomeada, o qual deve ser único entre todos os nomes de objetos, classes, métodos e funções do esquema. Caso o identificador tenha o mesmo nome de uma outra consulta nomeada existente, a nova definição substitui a definição anterior. Uma vez definida, a consulta especificada torna-se persistente até que seja redefinida ou removida. Uma visão também pode possuir parâmetros (argumentos) em sua definição.

Por exemplo, a visão VI a seguir define uma consulta nomeada *possui_al* unos para recuperar o conjunto dos objetos do tipo *aluno* que se especializam em um determinado departamento:

```
VI:  define possui_alunos(nomedepto) as select s
from s in alunos where s.estuda_em.dnome = nomedepto;
```

Como o esquema ODL da Figura 21.6 define um atributo unidirecional — *estuda_em* para *Al* uno, podemos utilizar a visão anterior para representar seu inverso sem ter que definir um relacionamento explicitamente. Esse tipo de visão pode ser utilizado para representar os relacionamentos inversos que não possuem expectativa de uso freqüente. O usuário pode, então, utilizar a visão anterior para escrever consultas como

```
possui_alunos('Ciências da Computação');
```

que deve retornar uma *bag* de *alunos* que estudam no departamento de ciências da computação. Observe que na Figura 21.6 definimos *forma_especialistas* como um relacionamento explícito, provavelmente porque é esperado que seja utilizado com maior freqüência.

Extrair Elementos Únicos de Coleções Unitárias. Em geral, uma consulta em OQL irá retornar uma coleção como seu resultado, como uma *bag*, um *set* (se *distinct* for especificado) ou uma *lista* (se for utilizada a cláusula *order by*). Se o usuário exigir que uma consulta retorne um único elemento, existe um operador de *element* na OQL que garante o retorno desse elemento único e de uma coleção unitária *c* que contém somente um elemento. Se *c* contiver mais que um elemento ou se *c* estiver vazia, o operador *element* causa uma exceção. Por exemplo, Q6 retorna a única referência de objeto para o departamento de ciência da computação:

```
Q6:  element (select d
from d in departamentos
where d.dnome = 'Ciências da Computação');
```

Uma vez que um nome de departamento é único para todos os departamentos, o resultado deve ser um único departamento. O tipo do resultado é *d: Departamento*.

Operadores de Coleções (Funções de Agregação, Quantificadores). Uma vez que muitas expressões de consulta especificam coleções como seus resultados, foi definido um conjunto de operadores que são aplicáveis em coleções. Esse conjunto inclui operadores de agregação, assim como associação e quantificação (universal e existencial) sobre uma coleção.

Os operadores de agregação (*min*, *max*, *count*, *sum* e *avg*) operam sobre uma coleção. O operador *count* retorna um tipo inteiro. Os demais operadores de agregação (*min*, *max*, *sum* e *avg*) retornam o mesmo tipo dos elementos da coleção. Seguem dois exemplos. A consulta Q7 retorna o número de alunos que estudam 'Ciências da Computação', enquanto Q8 retorna a média *mdc* de todos os alunos de nível superior que se especializam em Ciências da Computação.

```
Q7:  count (s in estuda_em('Ciências da Computação')); Q8: avg (select s.mdc
from s in alunos
```

25 Correspondem às funções de agregação em SQL.

496 Capítulo 21 Padrões, Linguagens e Projeto de Banco de Dados de Objetos

where s.especializa_em.dnome = 'Ciências da Computação' and s.turma = 'superior');

Observe que é possível aplicar operações de agregação a qualquer coleção com tipo compatível e pode-se utilizá-las em qualquer parte de uma consulta. Por exemplo, a consulta para recuperar todos os nomes de departamentos que possuem mais que 100 especialistas pode ser escrita como em Q9:

Q9: **select** d.dnome

from d **in** departamentos

where count (d.possui_especialistas) > 100;

As expressões de *pertinência* e *quantificação* retornam um tipo Booleano — verdadeiro ou falso. Seja v uma variável, c uma expressão de coleção, b uma expressão do tipo Booleano (ou seja, uma condição lógica) e e um elemento do tipo dos elementos da coleção c. Então:

(e in c) retorna verdadeiro se o elemento e for um membro da coleção c.

(for ai 1 v in c:b) retorna verdadeiro se todos os elementos da coleção c satisfizerem b.

(exists v in c:b) retorna verdadeiro se existir pelo menos um elemento em c que satisfaça b.

Para ilustrar a condição de pertinência, suponha que desejemos recuperar os nomes de todos os alunos que completaram o curso chamado 'Sistemas de Bancos de Dados I'. Isso pode ser escrito como em Q10, na qual a consulta aninhada retorna a coleção de nomes dos cursos que cada aluno s completou e a condição de pertinência retorna verdadeiro se 'Sistemas de Bancos de Dados I' estiver na coleção de determinado aluno s:

Q10: **select** s.nome.unome, s.nome.pnome **from** s **in** alunos

where 'Sistemas de Bancos de Dados I' **in** (select c.nome **from** c **in** s.disciplinas_cursadas.disciplina.do_curso);

Q10 também mostra um modo mais simples de se especificar a cláusula select de consultas que retornam uma coleção de estruturas; o tipo retornado por Q10 é uma bag<struct(string, string)>.

Também é possível escrever consultas que retornam resultados verdadeiro/falso. Como exemplo, vamos assumir que exista um objeto chamado Jeremy do tipo Aluno. Então, a consulta Q11 responde à seguinte pergunta: "Jeremy se especializa em Ciências da Computação?" Analogamente, Q12 responde à pergunta: "Todos os alunos de Ciências da Computação são orientados por professores do departamento de Ciências da Computação?". Tanto Q11 quanto Q12 retornam verdadeiro ou falso, que são interpretadas como respostas sim ou não para as perguntas anteriores:

Q11: Jeremy **in** especializa_em('Ciências da Computação'); Q12: **for ali g in** (select s

from s **in** gradaluno

where s.especializa_em.dnome = 'Ciências da Computação') : g.orientado_por **in** departamentocc.trabalha_em;

Observe que a consulta Q12 também mostra como a herança de atributos, relacionamento e operações é aplicada em consultas.

Embora s seja um iterator que percorre grad_aluno, podemos escrever s.especi ai i za_em porque o relacionamento especializa_em é herdado por Grad_Aluno de Aluno via EXTENDS (Figura 21.6). Finalmente, para ilustrar o quantificador exists, a consulta Q13 responde à seguinte pergunta: "Algum aluno de Ciências da Computação possui a mdc (média das disciplinas cursadas) igual a 4,0?"

Novamente, aqui, a operação mdc é herdada por Grad_Aluno de Aluno via EXTENDS.

Q13: **exists g in** (select s

from s **in** grad_aluno

where s.especializa_em.dnome = 'Ciências da Computação') : g.mdc = 4;

21.3 A Linguagem de Consulta de Objetos (OQL)

497

Expressões de Coleção Ordenadas (Indexadas). Como discutimos na Seção 21.1.2, coleções que sejam listas ou *arrays* têm operações adicionais, como a recuperação do *i*-ésimo, do primeiro e do último elementos. Além disso, existem operações para extrair uma subcoleção e concatenar duas listas. Assim, as expressões de consultas que envolvem listas ou *arrays* podem invocar essas operações. Ilustraremos algumas dessas operações utilizando consultas como exemplo. Q14 recupera o sobrenome do professor que ganha o maior salário:

Q14: **first (select** struct(professor: f.nome.unome, salário: f.salário) **from** f **in** professor **order by** f.salário **desc**);

Q14 ilustra a utilização do operador *first* em uma coleção list que contém os salários dos professores em ordem decrescente de salário. Dessa forma, o primeiro elemento dessa lista ordenada contém o professor com o maior salário. Essa consulta considera que somente um membro da faculdade ganha o salário máximo. A consulta seguinte, Q15, recupera os três melhores alunos que se especializam em Ciências da Computação, baseada na mdc.

Q15: **(select struct(sobrenome: s.nome.unome, primeiro_nome: s.nome.pnome, mdc: s.mdc) from s in departamentocc.forma_especialistas order by mdc desc) [0:2];**

A consulta select-from-order-by retorna uma lista de alunos de Ciências da Computação ordenados por mdc em ordem decrescente. O primeiro elemento de uma coleção ordenada possui uma posição de índice 0 (zero), de modo que a expressão [0:2] retorna uma lista contendo o primeiro, o segundo e o terceiro elementos do resultado da select-from-order-by.

O Operador de Agrupamento. A cláusula group by da OQL, embora semelhante à cláusula correspondente da SQL, fornece uma referência explícita à coleção de objetos em cada grupo ou *partição*. Primeiro, daremos um exemplo, e em seguida descreveremos a forma geral dessas consultas.

Q16 recupera o número de especialistas para cada departamento. Nessa consulta, os alunos são agrupados na mesma partição (grupo) se tiverem a mesma especialização; ou seja, o mesmo valor para s.especializa_em.dnome:

Q16: **select struct(nomedepcto, numero_de_especialistas: count (partition)) from s in alunos group by nomedepcto: s.especializa_em.dnome;**

O resultado da especificação do agrupamento é do tipo set<struct (nomedepcto: string, partition: bag<struct(s: Aluno)>>), que contém uma estrutura para cada grupo (PARTITION) com dois componentes: o valor do atributo de agrupamento (nomedepcto) e a bagde objetos de aluno no grupo (partition). A cláusula select retorna o atributo de agrupamento (nome do departamento) e uma contagem do número de elementos em cada partição (ou seja, o número de alunos em cada departamento), na qual partition é a palavra-chave utilizada para se referir a cada partição. O tipo do resultado da cláusula select é set<struct (nomedepcto: string, numero_de_especializacoes: integer)>. Em geral, a sintaxe para a cláusula group by é

group by f¹ e₁ f₂: e₂...f_k: e_k

em que f_j: e_j, f₂: e₂...f_k: e_k é uma lista de atributos de particionamento (agrupamento) e cada especificação de atributo de particionamento f_j: e_j define um nome de atributo (campo) f_j e uma expressão e_j. O resultado da aplicação de agrupamento (especificado na cláusula group by) é um conjunto de estruturas:

set<struct(f₁: t₁, f₂: t₂, ..., f_k: t_k, partition: bag>

na qual t_i é o tipo retornado pela expressão e_i, partition é um nome de campo discriminado (uma palavra-chave) e B é uma estrutura cujos campos são a variável de iteração (s em Q16) declarada na cláusula from que possui um tipo compatível.

Como na SQL, uma cláusula *having* pode ser utilizada para filtrar os conjuntos particionados (isto é, selecionar somente alguns grupos com base nas condições de grupo). Em Q17, a consulta anterior é modificada para ilustrar a cláusula having (e também mostrar a sintaxe simplificada para a cláusula select). Q17 recupera, para cada departamento que tenha mais de 100 alunos se especializando, a média da mdc de seus alunos. A cláusula having em Q17 seleciona somente as partições (grupos) que tenham mais de 100 elementos (ou seja, departamentos com mais de 100 alunos).

498 Capítulo 21 Padrões, Linguagens e Projeto de Banco de Dados de Objetos

Q17: **select** *nomedepto*, *mediajndc*: **avg(select p.s.mdc from p**
in partition) from *s* **in** *alunos*

group by *nomedepto*: *s.especializa_em.dnome* **having** **count (partition) > 100;**

Observe que a cláusula *select* de Q17 retorna a média da mdc dos alunos na partição. A expressão *select p.s.mdc from p in partition* retorna uma bag com as mdcs dos alunos na partição. A cláusula *from* declara uma variável de iteração *p* sobre a coleção de partições, que é do tipo *bag<struct (s: Aluno)>*. Então, a expressão de caminho *p.s.mdc* é utilizada para acessar a mdc de cada aluno na partição.

21.4 VISÃO GERAL DO BINDING COM A LINGUAGEM C++

O *binding* com a linguagem C++ especifica como as estruturas em ODL são mapeadas para estruturas em C++. Isso é realizado por uma biblioteca C++ que fornece as classes e as operações que implementam as estruturas em ODL. Uma linguagem de manipulação de objetos (OML) é necessária para especificar como os objetos do banco de dados são recuperados e manipulados dentro de um programa C++, tendo como base a sintaxe e a semântica da linguagem de programação C++. Além dos *bindings* ODL/OML, um conjunto de estruturas denominado *pragmas físico* (*physical pragmas*) é definido para permitir ao programador algum controle envolvendo questões associadas ao armazenamento físico, como *clustering* de objetos, atualização de índices e gerenciamento de memória.

A biblioteca de classes para o padrão ODMG adicionada ao C++ utiliza o prefixo *d_* para declarações de classes que se referem a conceitos de bancos de dados. O objetivo é que o programador deva pensar que somente uma linguagem está sendo utilizada, e não duas linguagens distintas. Uma classe *d_Ref<T>* é definida para cada classe *T* do esquema do banco de dados para o programador referenciar-se a objetos do banco de dados em um programa. Desse modo, as variáveis de programa do tipo *d_Ref<T>* podem se referir tanto a objetos persistentes como a transientes da classe *T*.

Para utilizar os vários tipos embutidos do modelo de objetos ODMG, como os tipos coleção, várias classes *templates* são especificadas na biblioteca. Por exemplo, uma classe abstrata *d_Object<T>* define as operações que são herdadas por todos os objetos. Analogamente, uma classe abstrata *d_Collection<T>* especifica as operações para coleções. Essas classes não são instanciáveis, mas apenas especificam as operações que podem ser herdadas por todos os objetos e por objetos de coleção, respectivamente. Uma classe *template* é especificada para cada tipo de coleção; ela inclui *d_Set<T>*, *d_List<T>*, *d_Bag<T>*, *d_Varray<T>* e *d_Dictionary<T>* e corresponde aos tipos de coleção do Modelo de Objetos (Seção 21.1). Assim, o programador pode criar classes de tipos como *d_Set<d_Ref<Aluno>*», cujas instâncias podem ser conjuntos de referências a objetos *Aluno*, ou *d_Set<String>* cujas instâncias podem ser conjuntos de cadeias de caracteres. Além disso, uma classe *d_Iterator* corresponde à classe *Iterator* do Modelo de Objetos.

A ODL C++ permite que um usuário especifique as classes de um esquema de banco de dados utilizando as estruturas de C++, assim como as estruturas fornecidas pela biblioteca de banco de dados de objetos. Para especificar os tipos de dados para atributos, são disponibilizados tipos básicos como *d_Short* (*short integer*), *d_JShort* (*unsigned short integer*), *d_Long* (*long integer*) e *d_Float* (número de ponto flutuante). Além dos tipos básicos de dados, vários tipos literais estruturados são disponibilizados e correspondem aos tipos literais estruturados do Modelo de Objetos ODMG. Esses tipos incluem *d_String*, *d_Interval*, *d_Date*, *d_Time* e *d_JTimestamp* (Figura 21.1b).

Para especificar relacionamentos, a palavra-chave *Rel_* é utilizada no prefixo dos nomes de tipos; por exemplo, ao escrevermos *d_Rel_Ref<Departamento, _possui_especialistas> especializa_em;* na classe *Aluno* e

d_Rel_Set<Aluno, especializa_em> possui_especialistas;

26 Provavelmente, *d_* representa classes de bancos de dados.

27 Isto é, *variáveis membros* na terminologia de programação orientada a objetos.

21.5 Projeto Conceitual de Banco de Dados de Objetos

499

na classe Departamento, estaremos declarando que *especifica* e *possui* são propriedades de relacionamentos que são inversas uma da outra e, dessa forma, representam um relacionamento binário 1:N entre Departamento e Aluno.

Para a OML (Linguagem de Manipulação de Objetos), o *binding* sobrecarrega a operação *new* de modo que ela possa ser utilizada para criar objetos persistentes ou transientes. Para criar objetos persistentes, deve ser fornecido o nome do banco de dados e o nome persistente do objeto. Por exemplo, ao escrever

```
d_Ref<Aluno> s = new(DB1,'John Smith') Aluno;
```

o programador cria um objeto persistente nomeado do tipo Aluno no banco de dados DB1 com nome persistente John_Smith. Uma outra operação, *delete_object()*, pode ser utilizada para remover objetos. A modificação de objetos é realizada por operações (métodos) definidos em cada classe pelo programador.

O *binding* com C++ também permite a criação de extensões com o uso da biblioteca de classes *d_Extent*. Por exemplo, ao escrever

```
d_Extent<Pessoa> TodasPessoas(DB1);
```

o programador criaria uma coleção de objetos nomeada como TodasPessoas — cujo tipo seria *d_Set<Pessoa>* — no banco de dados DB1, que mantém objetos persistentes do tipo Pessoa. No entanto, as restrições de chaves não são suportadas no *binding* com C++, e qualquer verificação de chave deve ser feita pelo programador por meio de métodos de classe. Além disso, o *binding* com C++ não permite persistência por alcançabilidade: o objeto deve ser declarado estaticamente para se tornar persistente no momento em que é criado.

21.5 PROJETO CONCEITUAL DE BANCO DE DADOS DE OBJETOS

A Seção 21.5.1 discute como o projeto de um banco de dados de objetos (BDO) diferencia-se do projeto de um banco de dados relacionais (BDR). A Seção 21.5.2 descreve um algoritmo de mapeamento que pode ser utilizado para criar um esquema de um BDO, composto pelas definições de classes em ODL ODMG, a partir de um esquema conceitual EER.

21.5.1 Diferenças entre o Projeto Conceitual de um BDO e de um BDR

Uma das principais diferenças entre o projeto de um BDO e um BDR é como os relacionamentos são tratados. Em um BDO, normalmente os relacionamentos são tratados pela definição de propriedade(s) de relacionamento ou atributo(s) de referência que inclui(em) o(s) OID(s) do(s) objeto(s) relacionado(s). Estes podem ser considerados como *referências a OIDs* para objetos relacionados. Tanto referências simples como coleções de referências são válidas. As referências para relacionamentos binários podem ser declaradas em uma única direção ou em ambas, dependendo do tipo de acesso desejado. Se forem declaradas em ambas as direções, podem ser especificados como inversos um do outro, estabelecendo em um BDO o equivalente à restrição de integridade referencial relacional.

Em um BDR, os relacionamentos entre tuplas (registros) são especificados por atributos com coincidência de valores. Estes podem ser considerados como *referências de valores* e são especificados por *chaves estrangeiras*, que são valores de atributos de chaves primárias repetidos em tuplas da relação em referência. Esses atributos estão limitados a valores simples em cada registro, pois atributos multivalorados não são permitidos no modelo relacional básico. Portanto, relacionamentos M:N devem ser representados não de forma direta, mas como uma relação (tabela) adicional, conforme discutido na Seção 7.1.

O mapeamento de relacionamentos binários que contêm atributos não é direto em BDOs, uma vez que o projetista deve escolher em quais direções os atributos de referência devem ser incluídos. Se os atributos são incluídos em ambas as direções, haverá redundância no armazenamento e isso pode levar a inconsistência de dados. Portanto, algumas vezes é preferível utilizar a abordagem relacional de se criar uma tabela adicional definindo-se uma classe separada para representar o relacionamento. Essa abordagem também pode ser utilizada para relacionamentos n-ários, com grau $n > 2$.

Outro ponto relevante nas diferenças de projeto de BDO e BDR é como a herança é tratada. Em BDO, essas estruturas estão embutidas no modelo, assim o mapeamento é realizado utilizando os construtores de herança, como *derived* (:) e EXTENDS. No projeto relacional, como foi discutido na Seção 7.2, há várias opções de escolha, uma vez que não há nenhuma estrutura predefinida para herança no modelo relacional básico. É importante salientar, porém, que os sistemas objeto-relacional e relacional estendido estão acrescentando características para se modelar diretamente essas estruturas, assim como para incluir especificações de operações em tipos abstratos de dados (Capítulo 22).

28 Fornecemos apenas uma breve visão geral do *binding* com C++. Para detalhes complementares, ver Cattel *et al.* (1997), Capítulo 5.

A terceira grande diferença é que no projeto de um BDO é necessário especificar as operações antecipadamente, um; vez que fazem parte das especificações das classes. Apesar da importância de se especificar operações durante a fase de projeto para todos os tipos do banco de dados, isso pode ser postergado no projeto relacional, pois não é rigorosamente exigido até a fase de implementação.

21.5.2 Mapeando um Esquema EER para um Esquema BDO

H relativamente automático projetar as declarações de tipos de classes de objetos para um SGBDO a partir de um esquema EEP que não contém categorias nem relacionamentos n-ários com $n > 2$. Contudo, as operações das classes não são especificadas no diagrama EER e devem ser adicionadas às declarações de classe após o término do mapeamento estrutural. As linhas gerais de mapeamento do EER para ODL são as seguintes:

Passo 1: Crie uma classe ODL para cada tipo de entidade ou subclasse do modelo EER. O tipo da classe ODL deve incluir todos os atributos da classe EER. Os atributos multivalorados são declarados utilizando-se os construtores *set*, *bag* ou *list*. Caso os valores em um atributo multivalorado de um objeto devam estar ordenados, o construtor *list* é escolhido; caso sejam permitidos valores duplicados, escolha-se o construtor *bag*; caso contrário, o construtor *set* é selecionado. Os atributos compostos são mapeados para um construtor de tuplas (utilizando-se a declaração *struct* em ODL).

Declare uma extensão para cada classe e especifique todos os atributos-chave como chaves da extensão (isso somente é possível se mecanismos para declarações de extensões e restrições de chaves forem disponíveis no SGBDO).

Passo 2: Adicione propriedades de relacionamentos ou atributos de referência para cada relacionamento binário nas classes ODL que participam do relacionamento. Estes podem ser criados em uma ou em ambas as direções. Caso um relacionamento binário seja representado por referências em ambas as direções, declare as referências como sendo propriedades do relacionamento que são inversas uma da outra, se tal mecanismo existir. Se um relacionamento binário é representado por uma referência em apenas uma direção, declare a referência como sendo um atributo da classe de referência cujo tipo é o nome da classe referenciada.

Dependendo do valor da cardinalidade do relacionamento binário, as propriedades do relacionamento ou atributos de referência podem ser monovalorados ou de tipos coleção. Serão monovalorados para relacionamentos binários nas direções 1:1 ou N:1; serão tipos coleção (valores de conjunto ou de lista) para relacionamentos nas direções 1:N ou M:N. Um modo alternativo para mapear relacionamentos binários M:N é discutido no Passo 7 a seguir.

Caso existam atributos do relacionamento, um construtor de tupla (*struct*) pode ser utilizado para criar uma estrutura da forma <referência, atributos do relacionamento>, que pode ser incluído no lugar do atributo de referência. No entanto, essa estrutura não permite o uso da restrição inversa. Além disso, se essa escolha for representada em ambas as direções, os valores de atributos serão representados duas vezes, criando redundância.

Passo 3: Inclua as operações pertinentes a cada classe. Estas não estão disponíveis no esquema EER e devem ser adicionadas ao projeto do banco de dados em consonância com os requisitos iniciais. Um método construtor deve incluir o código de programa que verifica quaisquer restrições que devem ser asseguradas quando um novo objeto é criado. Um método destrutor deve verificar as restrições que podem ser violadas quando um objeto é removido. Outros métodos devem incluir toda verificação de restrição adicional que seja relevante.

Passo 4: Uma classe ODL que corresponda a uma subclasse do esquema EER herda (por meio de EXTENDS) o tipo e os métodos de sua superclasse no esquema ODL. São especificados os seus atributos particulares (não herdados), referências de relacionamentos e operações, conforme discutidos nos passos 1, 2 e 3.

29

Isso implicitamente utiliza um construtor de tuplas no nível mais alto da declaração de tipo, mas em geral o construtor de tuplas não é explicitamente mostrado nas declarações de classes ODL.

E necessária uma análise adicional do domínio da aplicação para decidir qual construtor utilizar, pois essa informação não está disponível no esquema EER.

O padrão ODL fornece suporte para definições explícitas de relacionamentos inversos. Alguns produtos de SGBDOs podem não fornecer esse suporte; em tais casos, os programadores devem dar manutenção em todos os relacionamentos explicitamente, codificando os métodos necessários para atualizar os objetos. 32 A decisão de quando utilizar conjunto ou lista não está disponível a partir do esquema EER e deve ser determinada a partir dos requisitos.

30

51

21.6. de outros relacionamentos, a não ser o seu relacionamento de identidade: elas podem ser mapeadas como se fossem atributos compostos multivalorados do tipo de entidade

Resumo 501 propriedade, utilizando os construtores *set* ou *struct* ou *list*. Os atributos da entidade fraca são incluídos no construtor *struct*, que corresponde ao construtor de tuplas. Os atributos são mapeados conforme discutido nos passos 1 e 2.

Passo 5: Os tipos de entidades fracas podem ser mapeados do mesmo modo como tipos de entidades regulares. É possível um mapeamento alternativo para tipos de entidades fracas que não participam

Passo 6: As categorias (tipos *union*) de um esquema EER são difíceis de ser mapeadas para ODL. É possível criar um mapeamento semelhante ao mapeamento EER-para-relacional (Seção 7.2) pela declaração de uma classe para representar a categoria e definir relacionamentos 1:1 entre a categoria e cada uma de suas superclasses. Outra opção é utilizar um tipo *union*, se estiver disponível.

Passo 7: Um relacionamento n-ário com grau $n > 2$ pode ser mapeado em uma classe adicional com as referências correspondentes a cada classe participante. Essas referências são baseadas no mapeamento de um relacionamento 1:N de cada classe que representa um tipo de entidade participante para a classe que representa o relacionamento n-ário. Se preferir, um relacionamento binário M:N também pode utilizar essa opção de mapeamento, especialmente se possuir atributos de relacionamento.

O mapeamento foi aplicado a um subconjunto do esquema de banco de dados UNIVERSIDADE da Figura 4-10 no contexto do padrão de banco de dados de objetos ODMG. O esquema de objetos mapeados utilizando a notação ODL é mostrado na Figura 21.6.

21.6. RESUMO

Neste capítulo discutimos o padrão proposto para bancos de dados orientados a objetos. Iniciamos pela descrição das várias estruturas do modelo de objetos ODMG. Os diversos tipos embutidos, como *Object*, *Collection*, *Iterator*, *Set*, *List*, assim por diante, foram descritos pelas suas interfaces, que especificam as operações predefinidas de cada tipo.

Esses tipos embutidos formam a base sobre a qual a linguagem de definição de objetos (ODL) e a linguagem de consulta a objetos (OQL) se apoiam. Também descrevem os as diferenças entre objetos, que possuem um ObjectId, e literais, que correspondem em a valores sem OID. Os usuários podem declarar classes em suas aplicações que herdam operações de determinadas interfaces embutidas. Dois tipos de propriedades podem ser especificados em classes definidas pelo usuário — atributos e relacionamentos — além das operações que podem ser aplicadas aos objetos da classe. A ODL permite aos usuários especificar tanto interfaces como classes, e suporta

dois tipos de herança — herança de interface com uso de ":" e herança de classe por meio de EXTENDS. Uma classe pode possuir extensões e chaves.

Uma descrição da ODL foi apresentada e um exemplo de esquema de banco de dados para a base de dados UNIVERSIDADE foi empregado para ilustrar as estruturas da ODL. Na sequência, foi apresentada uma visão geral da linguagem de consulta a objetos (OQL). A OQL segue o princípio da ortogonalidade nas estruturas de consultas, significando que uma operação pode ser aplicada sobre o resultado de outra operação enquanto o tipo do resultado é o tipo correto de entrada para a operação. A sintaxe da OQL acompanha a maioria das estruturas da SQL, além de incluir conceitos adicionais, como expressões de caminho, herança, métodos, relacionamentos e coleções. Foram mostrados exemplos de como utilizar a OQL no banco de dados UNIVERSIDADE.

Apresentamos a seguir a visão geral do *bindingcom* a linguagem C++, que estende as declarações de classes C++ com os tipos construtores da ODL, além de permitir uma integração transparente de C++ com o SGBDO.

Após a descrição do modelo ODMG, apresentamos uma técnica de uso geral para projetar esquemas de bancos de dados orientados a objetos. Discutimos como os bancos de dados orientados a objetos diferenciam-se dos bancos de dados relacionais em três principais pontos: referências para representação de relacionamentos, inclusão de operações e herança. Mostramos como mapear um projeto conceitual de banco de dados no modelo EER para estruturas de bancos de dados de objetos.

Questões para Revisão

- 21.1. Quais são as diferenças e similaridades entre objetos e literais no modelo de objetos ODMG?
- 21.2. Liste as operações básicas das seguintes interfaces embutidas do modelo de objetos ODMG: *Object*, *Collection*, *Iterator*, *Set*, *Bag*, *Array* e *Dictionary*.
- 21.3. Descreva os literais estruturados predefinidos do modelo de objetos ODMG e as operações para cada um deles.

- 21.4- Quais são as diferenças e as semelhanças entre atributos e propriedades de relacionamentos em uma classe def pelo usuário (atômica)?
- 21.5. Quais são as diferenças e semelhanças entre herança de classes (EXTENDS) e de interface (":")?
- 21.6. Discuta como a persistência é especificada no *binding* com C++ do modelo de objetos ODMG.
- 21.7. Por que os conceitos de extensões e chaves são importantes em aplicações de bancos de dados?
- 21.8. Descreva os seguintes conceitos da OQL: *pontos de entrada no banco de dados, expressões de caminho, variáveis iteração, consultas nomeadas (visões), funções agregadas, agrupamentos e quantificadores*.
- 21.9. Qual o significado de ortogonalidade de tipos na OQL?
- 21.10. Discuta os princípios gerais do *binding* com a linguagem C++ no padrão ODMG.
- 21.11. Quais são as principais diferenças entre o projeto de um banco de dados relacionai e o de um banco de dados de obje
- 21.12. Descreva os passos do algoritmo para mapeamento EER para OO em projeto de banco de dados de objetos.

Exercícios

- 21.13. Projete um esquema OO para uma aplicação de banco de dados na qual você esteja interessado. Inicialmente devolva o esquema EER para a aplicação; a seguir, crie as classes correspondentes em ODL. Especifique alguns métodos para cada classe e escreva consultas em OQL para sua aplicação de banco de dados.
- 21.14. Considere o banco de dados AEROPORTO descrito no Exercício 4-21. Especifique uma quantidade de operações/métodos que você imagina que possa ser adequada a essa aplicação. Especifique as classes ODL e os métodos para o banco de dados.
- 21.15. Mapeie o esquema ER COMPANHIA da Figura 3.2 para classes ODL. Inclua os métodos apropriados para cada classe.
- 21.16. Especifique as consultas em OQL para os exercícios dos capítulos 7 e 8 que se referem ao banco de dados COMPAK

Bibliografia Seleccionada

Cattell et al. (1997) descrevem o padrão ODMG 2.0 e Cattell et al. (1993) descrevem as versões anteriores do padrão. Vários livros apresentam a arquitetura CORBA — por exemplo, Baker (1996). Outras referências gerais a bancos de dados orientados a objetos foram dadas nas notas bibliográficas do Capítulo 20.

O sistema O2 é descrito em Deux et al. (1991) e Bancilhon et al. (1992), que incluem uma lista de referências para várias publicações que descrevem outros aspectos de O2. O Modelo de O2 foi formalizado por Velez et al. (1989). O sistema ObjectStore é descrito em Lamb et al. (1991). Fishman et al. (1987) e Wilkinson et al. (1990) discutem sobre o ÍRIS, um sistema orientado a objetos que foi desenvolvido nos laboratórios Hewlett-Packard. Maier et al. (1986) e Butterworth et al. (1991) descrevem o projeto do GEMSTONE. Um sistema OO que suporta arquiteturas abertas desenvolvido na Texas Instruments é descrito em Thompson et al. (1993). O sistema ODE desenvolvido nos laboratórios Bell/AT&T é descrito em Agrawal e Gani (1989). O sistema ORION desenvolvido no MCC é descrito em Kim et al. (1990). Morsi et al. (1992) descrevem um beta OO.