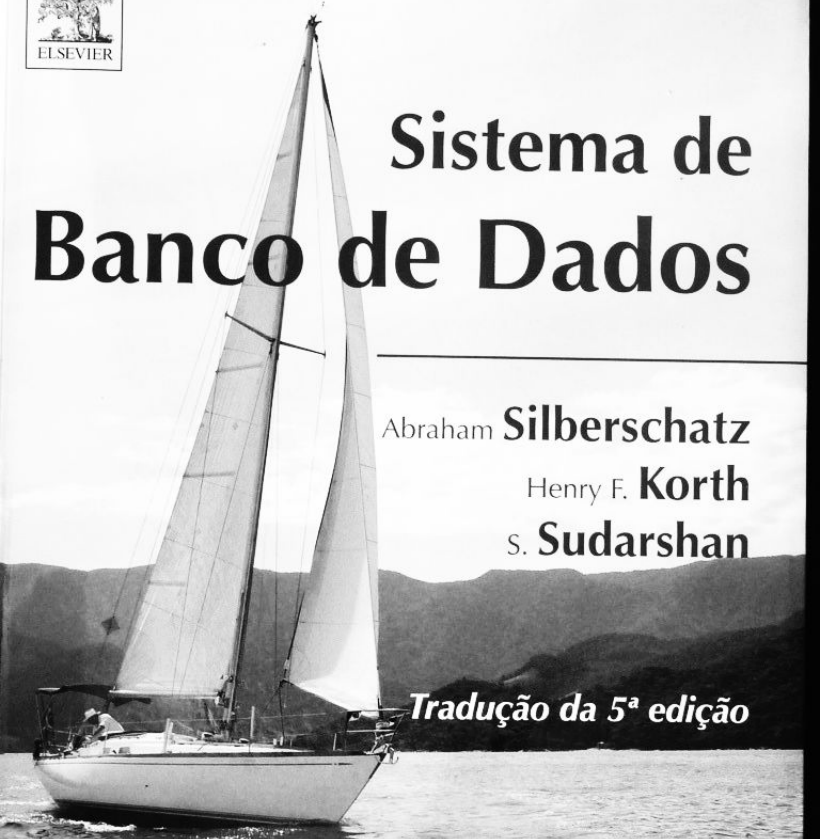
ELSEVIER

Sistema de Banco de Dados

Abraham **Silberschatz**
Henry F. **Korth**
S. **Sudarshan**

Tradução da 5ª edição



EDITORA CAMPUS

Material na WEB

Sistema de Banco de Dados

Abraham **Silberschatz**
Henry F. **Korth**
S. **Sudarshan**

Tradução da 5ª edição

Revisão Técnica

Luiz Fernando Pereira de Souza
Mestre em Engenharia de Sistemas pela UFRJ
Professor e Analista de Sistema do NCE/UFRJ

Tradução

Daniel Vieira
Presidente da Multinet Informática
Programador e tradutor especializado em Informática

ELSEVIER
EDITORA CAMPUS

Do original:
Database system concepts
Tradução autorizada do idioma inglês da edição publicada por McGraw-Hill – McGraw-Hill Companies, Inc.

Editoração Eletrônica
Estúdio Castellani

Revisão Gráfica
Marco Antonio Correa
Marilia Pinto de Oliveira

Copidesque:
Adriana Kramer

Projeto Gráfico
Elsevier Editora Ltda.
A Qualidade da Informação.
Rua Sete de Setembro, 111 – 16º andar
20050-006 Rio de Janeiro RJ Brasil
Telefone: (21) 3970-9300 FAX: (21) 2507-1991
E-mail: *info@elsevier.com.br*
Escritório São Paulo:
Rua Quintana, 753/8º andar
04569-011 Brooklin São Paulo SP
Tel.: (11) 5105-8555

ISBN 13: 978-85-352-1107-8
ISBN 10: 85-352-1107-8
Edição original: 0-07-295886-3

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual.
Em qualquer das hipóteses, solicitamos a comunicação à nossa Central de Atendimento, para que possamos esclarecer ou encaminhar a questão.
Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

Central de atendimento
Tel: 0800-265340
Rua Sete de Setembro, 111, 16º andar – Centro – Rio de Janeiro
e-mail: info@elsevier.com.br
site: www.campus.com.br

CIP-Brasil. Catalogação-na-fonte.
Sindicato Nacional dos Editores de Livros, RJ

S576s
Silberschatz, Abraham
Sistema de banco de dados / Abraham Silberschatz,
Henry F. Korth, S. Sudarshan ; tradução de Daniel Vieira.
– Rio de Janeiro : Elsevier, 2006.
il.

Tradução de: Database system concepts, 5th ed
Apêndice
Inclui bibliografia
ISBN 85-352-1107-8

1. Banco de dados – Gerência. I. Korth, Henry F. II.
Sudarshan, S. III. Título.

06-1033.

CDD 005.74
CDU 004.65

Em memória de meu pai Joseph Silberschatz,
minha mãe Vera Silberschatz
e meus avós Stepha e Aaron Rosenblum.
Avi Silberschatz

Para minha esposa, Joan,
meus filhos, Abigail e Joseph
e meus pais, Henry e Frances.
Hank Korth

Para minha esposa, Sita,
meu filho, Madhur,
e minha mãe, Indira.
S. Sudarshan

Agradecimentos

Esta edição se beneficiou com os muitos comentários úteis que nos foram oferecidos pelos diversos alunos que usaram as quatro edições anteriores. Além disso, muitas pessoas escreveram ou nos falaram a respeito do livro, oferecendo sugestões e comentários. Embora não possamos mencionar todas essas pessoas aqui, somos especialmente gratos aos seguintes:

- Hani Abu-Salem, DePaul University; Jamel R. Alsabbagh, Grand Valley State University; Ramzi Bualuan, Notre Dame University; Zhengxin Chen, University of Nebraska em Omaha; Jan Chomick, SUNY Buffalo University; Qin Ding, Penn State University em Harrisburg; Frantisek Franek, McMaster University; Shashi K. Gadia, Iowa State University; William Hankley, Kansas State University; Randy M. Kaplan, Drexel University; Mark Llewellyn, University of Central Florida; Marty Maskarinec, Western Illinois University; Yiu-Kai Dennis Ng, Brigham Young University; Sunil Prabhakar, Purdue University; Stewart Shen, Old Dominion University; Anita Whitehall, Foothill College; Christopher Wilson, University of Oregon; Weining Zhang, University of Texas em San Antonio; que serviram como revisores do livro e cujos comentários nos ajudaram muito na formulação desta quinta edição.
- Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou e Bianca Schroeder (Carnegie Mellon University), por escreverem o Apêndice descrevendo o sistema de banco de dados PostgreSQL.
- Hakan Jakobsson (Oracle), pelo Apêndice sobre o sistema de banco de dados Oracle.
- Sriram Padmanabhan (IBM), pelo Apêndice descrevendo o sistema de banco de dados IBM DB2.
- Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas e Michael Zwilling (todos da Microsoft) pelo Apêndice sobre o sistema de banco de dados Microsoft SQL Server. José Blakeley também pela coordenação e revisão do Capítulo 29 e César Galindo-Legaria, Goetz Graefe, Kalen Delaney e Thomas Casey (todos da Microsoft) por suas contribuições à edição anterior do capítulo sobre Microsoft SQL Server.
- Chen Li e Sharad Mehrotra por fornecer material sobre SQL e segurança que nos ajudou a atualizar e estender o Capítulo 8.
- Valentin Dinu, Goetz Graefe, Bruce Hillyer, Chad Hogg, Nahid Rahman, Patrick Schmid, Jeff Storey, Prem Thomas, Liu Zhenming e, particularmente, N. L. Sarda, por seu feedback, que nos ajudou a preparar a quinta edição.
- Rami Khouri, Nahid Rahman e Michael Rys, pelo feedback sobre as versões de rascunho dos capítulos da quinta edição.
- Raj Ashar, Janek Bogucki, Gavin M. Bierman, Christian Breimann, Tom Chappell, Y. C. Chin, Laurens Damen, Prasanna Dhandapani, Arvind Hulgeri, Zheng Jiaping, Graham J. L. Kemp, Hae Choon Lee, Sang-Won Lee, Thanh-Duy Nguyen, D. B. Phatak, Juan Altmayer Pizzorno, Rajarshi Rakshit, Greg Riccardi, N. L. Sarda, Max Smolens, Nikhil Sethi e Tim Wahls por apontar erros na quarta edição.
- Marilyn Turnamian, cujo excelente auxílio como secretária foi essencial para o término dessa quinta edição em tempo.

A editora foi Betsy Jones. A editora patrocinadora foi Kelly Lowery. A editora de desenvolvimento foi Melinda D.

Bilecki. A gerente de projeto foi Peggy Selle. O gerente executivo de marketing foi Michael Weitz. O gerente de marketing foi Dawn Bercier. A ilustração e o projeto da capa foram de JoAnne Schopler. O revisor de textos autônomo foi George Watson. A revisora de provas autônoma foi Judy Gantenbein. O projeto do livro foi de Laurie Janssen. O indexador autônomo foi Tobiah Waldron.

Esta edição é baseada nas quatro edições anteriores, de modo que agradecemos mais uma vez às muitas pessoas que nos ajudaram com as quatro primeiras edições, incluindo R. B. Abhyankar, Don Batory, Phil Bernhard, Haran Boral, Paul Bourgeois, Phil Bohannon, Robert Brazile, Yuri Breitbart, Michael Carey, Soumen Chakrabarti, J. Edwards, Christos Faloutsos, Homma Farian, Alan Fekete, Shashi Gadia, Jim Gray, Le Gruenwald, Eitan M. Gurari, Ron Hitchens, Yannis Ioannidis, Hyoung-Joo Kim, Won Kim, Henry Korth (pai de Henry F.), Carol Kroll, Gary Lindstrom, Irwin Levinstein, Ling Liu, Dave Maier, Keith Marzullo, Fletcher Mattox, Sharad Mehrotra, Jim Melton, Alberto Mendelzon, Hector Garcia-Molina, Ami Motro, Bhagirath Narahari, Anil Nigam, Cyril Orji, Meral Ozsoyoglu, Bruce Porter, Jim Peterson, K. V. Raghavan, Krithi Ramamritham, Mike Reiter, Odinaldo Rodriguez, Mark Roth, Marek Rusinkiewicz, Sunita Sarawagi, N. L. Sarda, S. Seshadri, Shashi Shekhar, Amit Sheth, Nandit Soparkar, Greg Speegle, Dilys Thomas e Marianne Winslett.

Marilyn Turnamian e Nandprasad Joshi ofereceram auxílio de secretariado para a quarta edição, e Marilyn também preparou um rascunho inicial do projeto da capa para a quarta edição. Lyn Dupré revisou a terceira edição e Sara Strandtman editou o texto da terceira edição. Nilesh Dalvi, Sumit Sanghai, Gaurav Bhalotia, Arvind Hulgeri K. V. Raghavan, Prateek Kapadia, Sara Strandtman, Greg Speegle e Dawn Bezviner ajudaram a preparar o manual do instrutor para as edições anteriores. A nova capa é uma evolução das capas das quatro primeiras edições. A idéia de usar barcos como parte do conceito da capa nos foi sugerida inicialmente por Bruce Stephan.

Finalmente, Sudarshan gostaria de agradecer sua esposa, Sita, por seu amor e apoio, e seu filho Madhur, por seu amor. Hank gostaria de agradecer sua esposa, Joan, e seus filhos, Abby e Joe, por seu amor e compreensão. Avi gostaria de agradecer a Valerie por seu amor, paciência e apoio durante a revisão deste livro.

A. S.
H. F. K.
S. S.

Prefácio

O gerenciamento de banco de dados evoluiu de uma aplicação de computador especializada para um componente central do ambiente de computação moderno e, como resultado, o conhecimento a respeito dos sistemas de banco de dados se tornou uma parte essencial de um curso de ciência de computação. Neste texto, apresentamos os conceitos fundamentais do gerenciamento de banco de dados. Esses conceitos incluem aspectos de projeto de banco de dados, linguagens de banco de dados e implementação de sistemas de banco de dados.

Este texto é voltado para um primeiro curso de banco de dados, em um nível de estudante universitário iniciante ou não, ou para o primeiro ano de pós-graduação. Além do material básico para um curso inicial, o texto contém material avançado que pode ser usado para suplementos de curso, ou como material introdutório para um curso avançado.

Consideramos apenas uma familiaridade com as estruturas de dados básicas, organização de computador e uma linguagem de programação de alto nível, como Java, C ou Pascal. Apresentamos os conceitos como descrições intuitivas, muitas delas baseadas em nosso exemplo comum de uma instituição bancária. Abordamos resultados teóricos importantes, mas as provas formais são omitidas. No lugar das provas, figuras e exemplos são usados para sugerir por que um resultado é verdadeiro. Descrições formais e provas dos resultados teóricos poderão ser encontradas nos trabalhos de pesquisa e textos avançados que são referenciados nas notas bibliográficas.

Os conceitos e algoritmos fundamentais abordados no livro freqüentemente são baseados naqueles utilizados nos sistemas de banco de dados existentes, tanto comerciais quanto experimentais. Nosso objetivo é apresentar esses conceitos e algoritmos em um ambiente geral, que não esteja ligado a um sistema de banco de dados em particular. Os detalhes dos sistemas de banco de dados em particular são discutidos na Parte 9.

Neste livro, a quinta edição do original de *Sistemas de banco de dados*, mantivemos o estilo geral das edições anteriores, enquanto evoluímos o conteúdo e a organização, a fim de refletir as mudanças que estão ocorrendo na forma como os bancos de dados são projetados, gerenciados e utilizados. Também levamos em consideração as tendências no ensino dos conceitos de banco de dados e fizemos adaptações para facilitar essas tendências, onde for apropriado. Antes de descrevermos o conteúdo do livro com detalhes, destacamos algumas das características da quinta edição.

- **Cobertura da SQL mais cedo.** Muitos instrutores utilizam SQL como um componente-chave dos projetos do semestre (veja alguns exemplos de projetos em nosso site, www.db-book.com). Para dar aos alunos bastante tempo para os projetos, principalmente para universidades e colégios em sistema de trimestre, é essencial ensinar SQL o mais cedo possível. Com isso em mente, realizamos várias mudanças na organização:
-
  1. Adiamos a apresentação do modelo entidade-relacionamento para a Parte 2.
  2. Agilizamos a introdução do modelo relacional, deixando a cobertura do cálculo relacional para o Capítulo 5, mantendo a cobertura da álgebra relacional no Capítulo 2.
  3. Dedicamos dois capítulos iniciais à SQL. O Capítulo 3 abrange os recursos básicos da SQL, incluindo definição e manipulação de dados. O Capítulo 4 aborda recursos mais avançados, incluindo restrições de

integridade, SQL dinâmica e construções de procedimento. O material novo nesse capítulo inclui uma cobertura expandida da SQL, construções de procedimento na SQL, recursão na SQL e novos recursos da SQL:2003. O capítulo também inclui uma rápida visão geral da autorização; a cobertura detalhada da autorização foi adiada para o Capítulo 8.

Essas mudanças permitem que os alunos comecem a escrever consultas SQL mais cedo no curso, ganhando familiaridade com o uso de sistemas de banco de dados. Isso também permite que os alunos desenvolvam uma intuição sobre projeto de banco de dados, o que facilita o ensino da metodologia de projeto na Parte 2 do texto. Descobrimos que os alunos apreciam melhor as questões de projeto de banco de dados com essa organização.

- **Uma nova parte (Parte 2)**, que é dedicada ao projeto de banco de dados. A Parte 2 do texto contém três capítulos dedicados ao projeto de bancos de dados e aplicações de banco de dados. Incluímos aqui um capítulo (Capítulo 6) sobre o modelo entidade-relacionamento, que inclui todo o material do capítulo correspondente da quarta edição (Capítulo 2), mais várias atualizações significativas. Também apresentamos no Capítulo 6 uma rápida introdução ao processo de projeto de banco de dados. Os instrutores que preferirem iniciar seu curso com o modelo E-R poderão começar com esse capítulo sem perda de continuidade, pois nos esforçamos muito para evitar dependências de qualquer capítulo anterior, a não ser pelo Capítulo 1.

  O Capítulo 7, sobre projeto relacional, apresenta o material incluído no Capítulo 7 da quarta edição, mas em um estilo novo, mais legível. Os conceitos de projeto do modelo E-R são usados para oferecer uma introdução intuitiva às questões do projeto relacional, em antecipação à apresentação da técnica formal de projeto, usando dependências funcionais e de valores múltiplos, e normalização algorítmica. Esse capítulo também inclui uma nova seção sobre questões temporais no projeto de banco de dados.

  A Parte 2 termina com um novo capítulo, o Capítulo 8, que descreve o projeto e o desenvolvimento de aplicações de banco de dados, incluindo aplicações Web, servlets, JSP, triggers e questões de segurança. Acompanhando a necessidade cada vez maior de proteger o software contra ataques, a cobertura sobre segurança foi significativamente expandida desde a quarta edição.
- **Cobertura totalmente revisada e atualizada sobre bancos de dados baseados em objeto e XML**. A Parte 3 inclui um capítulo totalmente revisado sobre bancos de dados baseados em objeto, que enfatiza os recursos de objeto relacional da SQL, substituindo os capítulos separados sobre bancos de dados orientados a objeto e objeto relacional da quarta edição. Parte do material introdutório sobre orientação a objeto, com os quais os alunos estão acostumados devido a cursos anteriores, foi removida, assim como detalhes sintáticos do padrão ODMG, agora extinto. Porém, conceitos importantes e fundamentais para os bancos de dados orientados a objeto foram retidos, incluindo o material novo sobre o padrão JDO, para inclusão de persistência à Java.

  A Parte 3 inclui também um capítulo sobre o projeto e a consulta de dados XML, o qual foi significativamente revisado a partir do capítulo correspondente na quarta edição. Ele inclui uma cobertura avançada sobre XMLSchema e XQuery, cobertura sobre o padrão SQL/XML e mais exemplos de aplicações XML, incluindo Web services.
- **Material reorganizado sobre data mining e recuperação de informações**. O data mining e o processamento analítico on-line (OLAP) agora são usos muito importantes dos bancos de dados – não mais apenas "tópicos avançados". Portanto, passamos nossa explicação desses tópicos para uma nova parte, a Parte 6, contendo um capítulo sobre data mining e análise e um capítulo sobre recuperação de informações.
- **Novo estudo de caso abordando PostgreSQL**. É um sistema de banco de dados de fonte aberto que obteve grande popularidade nos últimos anos. Além de ser uma plataforma para a montagem de aplicações de banco de dados, o código-fonte pode ser estudado e estendido nos cursos que enfatizam os detalhes internos do banco de dados. Portanto, um estudo de caso do PostgreSQL é incluído na Parte 9, quando se junta a três estudos de caso que aparecem na quarta edição (Oracle, IBM DB2 e Microsoft SQL Server). Os três últimos estudos de caso foram atualizados para refletir as versões mais recentes do respectivo software.

Todos os assuntos não listados aqui – incluindo o processamento de transações (concorrência e recuperação), estruturas de armazenamento, processamento de consulta e bancos de dados distribuídos e paralelos – foram atualizados a partir dos textos correspondentes na quarta edição, embora sua organização geral esteja relativamente inalterada. A cobertura sobre QBE no Capítulo 5 foi revisada, removendo os detalhes sintáticos da agregação e atualizações que não correspondem a qualquer implementação real, embora retendo os princípios conceitos por trás do QBE.

Organização

O texto está organizado em nove partes principais, mais três apêndices.

- **Visão geral** (Capítulo 1). O Capítulo 1 oferece uma visão geral da natureza e propósito dos sistemas de banco de dados. Explicamos como o conceito de um sistema de banco de dados se desenvolveu, quais são os recursos comuns dos sistemas de banco de dados, o que um sistema de banco de dados faz para o usuário e como um sistema de banco de dados realiza sua interface com os sistemas operacionais. Também introduzimos uma aplicação de banco de dados de exemplo: uma instituição bancária consistindo em várias filiais de banco. Esse exemplo é utilizado por todo o livro. O capítulo é motivador, histórico e explicativo por natureza.
- **Parte 1: Bancos de dados relacionais** (Capítulos de 2 a 5). O Capítulo 2 introduz o modelo relacional de dados, incluindo conceitos básicos e também a álgebra relacional. O capítulo também oferece uma rápida introdução às restrições de integridade. Os Capítulos 3 e 4 focalizam a mais influente das linguagens relacionais orientadas ao usuário: SQL. Enquanto o Capítulo 3 oferece uma introdução básica à SQL, o Capítulo 4 descreve recursos mais avançados da SQL, incluindo como realizar a interface entre uma linguagem de programação e uma SQL de suporte ao banco de dados. O Capítulo 5 aborda outras linguagens relacionais, incluindo o cálculo relacional, QBE e Datalog.

  Os capítulos dessa parte descrevem a manipulação de dados: consultas, atualizações, inserções e exclusões, assumindo que um projeto de esquema foi oferecido. As questões de projeto de esquema são deixadas para a Parte 2.
- **Parte 2: Projeto de banco de dados** (Capítulos de 6 a 8). O Capítulo 6 oferece uma visão geral do processo de projeto de banco de dados, com a ênfase principal no projeto de banco de dados usando o modelo de dados entidade-relacionamento. O modelo de dados entidade-relacionamento oferece uma visão de alto nível das questões de projeto de banco de dados e dos problemas que encontramos na captura da semântica de aplicações realistas dentro das restrições de um modelo de dados. A notação de diagrama de classes da UML também é abordada nesse capítulo.

  O Capítulo 7 apresenta a teoria do projeto de banco de dados relacional. A teoria das dependências funcionais e da normalização é abordada, enfatizando a motivação e o conhecimento intuitivo de cada forma normal. Esse capítulo começa com uma visão geral do projeto relacional e conta com um conhecimento intuitivo da implicação lógica das dependências funcionais. Isso permite que o conceito de normalização seja introduzido antes da cobertura completa da teoria da dependência funcional, que é apresentada mais adiante no capítulo. Os instrutores podem decidir usar apenas essa cobertura inicial nas Seções de 7.1 a 7.3, sem perda da continuidade. Os instrutores que utilizam o capítulo inteiro se beneficiarão com os alunos tendo um bom conhecimento dos conceitos de normalização, a fim de motivar alguns dos conceitos desafiadores da teoria da dependência funcional.

  O Capítulo 8 aborda o projeto e o desenvolvimento de aplicações. Esse capítulo enfatiza a construção de aplicações de banco de dados com interfaces baseadas na Web. Além disso, o capítulo aborda a segurança da aplicação.
- **Parte 3: Bancos de dados baseados em objeto e XML** (Capítulos 9 e 10). O Capítulo 9 aborda os bancos de dados baseados em objeto. O capítulo descreve o modelo de dados objeto-relacional, que estende o modelo de dados relacional para dar suporte aos tipos de dados complexos, herança de tipo e identidade de objeto. O capítulo também descreve o acesso ao banco de dados por meio de linguagens de programação orientadas a objeto.

  O Capítulo 10 aborda o padrão XML para representação de dados, cujo uso está cada vez maior na troca e armazenamento de dados complexos. O capítulo também descreve as linguagens de consulta para XML.
- **Parte 4: Armazenamento e consulta de dados** (Capítulos de 11 a 14). O Capítulo 11 trata da estrutura de disco, arquivo e do sistema de arquivos. Diversas técnicas de acesso aos dados são apresentadas no Capítulo 12, incluindo hashing e índices de árvore B+. Os Capítulos 13 e 14 enfatizam algoritmos de avaliação de consulta e otimização de consulta. Esses capítulos oferecem um conhecimento dos detalhes internos dos componentes de armazenamento e recuperação de um banco de dados.
- **Parte 5: Gerenciamento de transações** (Capítulos de 15 a 17). O Capítulo 15 focaliza os fundamentos do sistema de processamento de transações, incluindo atomicidade, consistência, isolamento e durabilidade da transação, além da noção de seriação.

  O Capítulo 16 focaliza o controle de concorrência e apresenta diversas técnicas para garantir a seriação, incluindo bloqueio, timestamping e técnicas otimistas (validação). O capítulo também aborda questões de impasse.

  O Capítulo 17 aborda as principais técnicas para garantir a execução correta da transação, apesar de panes do sistema e falhas de disco. Essas técnicas incluem logs, pontos de verificação e dumps de banco de dados.
- **Parte 6: Data mining e recuperação de informações** (Capítulos 18 e 19). O Capítulo 18 apresenta o conceito de um data warehouse e explica o data mining e o processamento analítico on-line (OLAP), incluindo suporte à SQL:1999 para OLAP e data warehousing. O Capítulo 19 descreve técnicas de recuperação de informações

para consulta de dados textuais, incluindo técnicas baseadas em hiperlink utilizadas nos mecanismos de pesquisa da Web.

A Parte 6 utiliza os conceitos de modelagem e linguagem das Partes 1 e 2, mas não depende das Partes 3, 4 ou 5. Portanto, ela pode ser facilmente incorporada em um curso que focaliza SQL e projeto de banco de dados.

- **Parte 7: Arquitetura de sistema de banco de dados** (Capítulos de 20 a 22). O Capítulo 20 aborda a arquitetura de sistemas de computador e descreve a influência do sistema de computador básico sobre o sistema de banco de dados. Discutimos sistemas centralizados, sistemas cliente-servidor, arquiteturas paralela e distribuída e tipos de rede nesse capítulo.

  O Capítulo 21, sobre bancos de dados paralelos, explora as diversas técnicas de paralelismo, incluindo paralelismo de E/S, paralelismo interconsulta e intraconsulta, e paralelismo interoperação e intra-operação. O capítulo também descreve o projeto de sistema paralelo.

  O Capítulo 22 aborda os sistemas de bancos de dados distribuídos, retornando às questões de projeto de banco de dados, gerenciamento de transações e avaliação e otimização de consulta, no contexto dos bancos de dados distribuídos. O capítulo também aborda questões de disponibilidade de sistema durante falhas e descreve o sistema de diretório LDAP.
- **Parte 8: Outros tópicos** (Capítulos de 23 a 25). O Capítulo 23 aborda os benchmarks de desempenho, ajuste de desempenho, padronização e migração de aplicações de sistemas legados.

  O Capítulo 24 aborda os tipos de dados avançados e novas aplicações, incluindo dados temporais, dados espaciais e geográficos, dados de multimídia e questões de gerenciamento de bancos de dados móveis e pessoais.

  Finalmente, o Capítulo 25 trata do processamento avançado de transações. Os tópicos abordados no capítulo incluem monitores de processamento de transação, fluxos de trabalho transacionais, comércio eletrônico, sistemas de transação de alto desempenho, sistemas de transação em tempo real, transações de longa duração e gerenciamento de transações em sistemas de múltiplos bancos de dados.
- **Parte 9: Estudos de caso** (Capítulos de 26 a 29). Nessa parte, apresentamos estudos de caso de quatro dos principais sistemas de banco de dados, incluindo PostgreSQL, Oracle, IBM DB2 e Microsoft SQL Server. Esses capítulos esboçam os recursos exclusivos de cada um desses sistemas, descrevendo sua estrutura interna. Eles oferecem diversas informações interessantes sobre os respectivos produtos, ajudando o leitor a ver como as diversas técnicas de implementação descritas nas partes anteriores são usadas em sistemas reais. Eles também abordam vários aspectos práticos interessantes no projeto de sistemas reais.
- **Apêndices on-line**. Embora a maioria das aplicações de banco de dados utilize o modelo relacional ou o modelo objeto-relacional, os modelos de dados de rede e hierárquico ainda estão em uso em algumas aplicações legadas. Para o benefício dos leitores que desejam aprender sobre esses modelos de dados, oferecemos apêndices descrevendo os modelos de dados de rede e hierárquico, nos Apêndices A e B, respectivamente; os apêndices estão disponíveis apenas on-line.

  O Apêndice C descreve o projeto avançado de banco de dados relacional, incluindo a teoria de dependências de múltiplos valores, dependências de junção e as formas normais de junção de projeto e chave de domínio. Esse apêndice beneficia aqueles indivíduos que desejam estudar a teoria do projeto de banco de dados relacional com mais detalhes, e instrutores que queiram fazer isso em seus cursos. Esse apêndice também está disponível apenas on-line.

A quinta edição

A produção desta quinta edição levou em consideração os muitos comentários e sugestões que recebemos com relação às edições anteriores, nossas próprias observações enquanto lecionávamos na Yale University, Lehigh University e IIT de Bombaim, e nossa análise das direções para onde a tecnologia de banco de dados está evoluindo.

Nosso procedimento básico foi reescrever o material de cada capítulo, atualizar o material mais antigo, acrescentar discussões sobre desenvolvimentos recentes na tecnologia de banco de dados e melhorar as descrições dos tópicos que os alunos acharam difíceis de entender. Assim como na quarta edição, cada capítulo possui uma lista de termos de revisão que poderá ajudar os leitores a rever os principais assuntos abordados no capítulo. A maioria dos capítulos também possui uma seção de ferramentas ao final do capítulo, que oferece informações sobre ferramentas de software relacionadas ao assunto do capítulo. Também acrescentamos novos exercícios e referências atualizadas.

Na quinta edição, dividimos os exercícios em dois conjuntos: **exercícios práticos** e **exercícios**. As soluções para os exercícios práticos estão disponíveis publicamente na página Web do livro. Os alunos são encorajados a solucionar os exercícios práticos por conta própria, e depois usar as soluções na página Web para comparar com suas próprias soluções. As soluções dos outros exercícios estão disponíveis apenas para os instrutores (veja em "Nota ao instrutor", a seguir, as informações sobre como obter as soluções).

Nota ao instrutor

O livro contém material básico e avançado, que pode não ser abordado em um único semestre. Marcamos várias seções como avançadas, usando o símbolo **. Essas seções poderão ser omitidas, se for desejado, sem perda de continuidade. Os exercícios que são difíceis (e que podem ser omitidos) também estão marcados com o símbolo "**".

É possível elaborar cursos usando vários subconjuntos dos capítulos. Esboçamos aqui algumas das possibilidades:

- As seções do Capítulo 4, da Seção 4.6 em diante, podem ser omitidas de um curso introdutório.
- O Capítulo 5 pode ser omitido se os alunos não forem utilizar cálculo relacional, QBE ou Datalog como parte do curso.
- Os Capítulos 9 (Bancos de dados baseados em objeto), 10 (XML) e 14 (Otimização de consulta) podem ser omitidos de um curso introdutório.
- Tanto nossa explicação sobre processamento de transações (Capítulos de 15 a 17) quanto nossa explicação sobre arquitetura de sistema de banco de dados (Capítulos de 20 a 22) consistem em um capítulo introdutório (Capítulos 15 e 20, respectivamente), seguido por capítulos com detalhes. Você poderia decidir usar os Capítulos 15 e 20, omitindo os Capítulos 16, 17, 21 e 22, se preferir deixar esses últimos capítulos para um curso avançado.
- Os Capítulos 18 e 19, que explicam sobre data mining e recuperação de informações, podem ser usados pelo aluno como material autodidata ou podem ser omitidos de um curso introdutório.
- Os Capítulos de 23 a 25 são adequados para um curso avançado ou para estudo individualizado dos alunos.
- Os capítulos de estudos de caso, de 26 a 29, são adequados para estudo individualizado dos alunos.

Sumário

Parte 7 Arquitetura do sistema

Parte 8 Outros tópicos

CAPÍTULO 1

Introdução

Um **sistema de gerenciamento de banco de dados** (DBMS) é uma coleção de dados inter-relacionados e um conjunto de programas para acessar esses dados. A coleção de dados, normalmente chamada de **banco de dados**, contém informações relevantes a uma empresa. O principal objetivo de um DBMS é fornecer uma maneira de recuperar informações de banco de dados que seja tanto *conveniente* quanto *eficiente*.

Os sistemas de banco de dados são projetados para gerenciar grandes blocos de informação. O gerenciamento de dados envolve definir estruturas para armazenamento de informação e fornecer mecanismos para a manipulação de informações. Além disso, o sistema de banco de dados precisa garantir a segurança das informações armazenadas, apesar das falhas de sistema ou de tentativas de acesso não autorizado. Se os dados precisarem ser compartilhados entre vários usuários, o sistema precisa evitar possíveis resultados anômalos.

Como as informações são muito importantes na maioria das organizações, os cientistas da computação desenvolveram um grande grupo de conceitos e técnicas para gerenciar dados. Esses conceitos e técnicas formam o foco deste livro. Este capítulo apresenta brevemente os princípios dos sistemas de banco de dados.

Aplicações do sistema de banco de dados

Os bancos de dados são amplamente usados. Aqui estão algumas aplicações representativas:

- *Banco*: para informações de cliente, contas, empréstimos e transações bancárias.
- *Linhas aéreas*: para reservas e informações de horários. As linhas aéreas foram umas das primeiras a usar bancos de dados de maneira geograficamente distribuída.
- *Universidades*: para informações de alunos, registros de cursos e notas.
- *Transações de cartão de crédito*: para compras com cartões de crédito e geração de faturas mensais.
- *Telecomunicação*: para manter registros de chamadas realizadas, gerar cobranças mensais, manter saldos de cartões de chamada pré-pagos e armazenar informações sobre as redes de comunicações.
- *Finanças*: para armazenar informações sobre valores mobiliários, vendas e compras de instrumentos financeiros como ações e títulos; também para armazenar dados de mercado em tempo real a fim de permitir negócios on-line por clientes e transações automatizadas pela empresa.
- *Vendas*: para informações de cliente, produto e compra.
- *Revendedores on-line*: para os dados de vendas descritos aqui, além de acompanhamento de pedidos, geração de listas de recomendação e manutenção de avaliações de produto on-line.
- *Indústria*: para gerenciamento da cadeia de suprimento e para controlar a produção de itens nas fábricas, estoques de itens em armazéns e lojas, além de pedidos de itens.
- *Recursos humanos*: para informações sobre funcionários, salários, descontos em folha de pagamento, benefícios e para geração de contracheques.

Como a lista mostra, os bancos de dados formam uma parte essencial de quase todas as empresas atuais.

Durante as últimas quatro décadas do século XX, o uso dos bancos de dados cresceu em todas as empresas. Nos

primeiros dias, muito poucas pessoas interagiam diretamente com os sistemas de banco de dados, embora o fizessem indiretamente, sem perceber – por meio de relatórios impressos, como faturas de cartão de crédito, ou de agentes, como caixas bancários e agentes de reserva de passagens aéreas. Então, as máquinas de auto-atendimento apareceram e deixaram os usuários interagirem diretamente com os bancos de dados. As interfaces telefônicas com computadores (sistemas de resposta de voz interativos) também permitiram que os usuários lidassem diretamente com bancos de dados – um chamador podia discar um número e pressionar teclas do telefone para inserir informações ou selecionar opções alternativas, para saber horários de chegada e saída, por exemplo, ou para se matricular em cursos de uma universidade.

A revolução da Internet no final da década de 1990 aumentou muito o acesso direto de usuários a bancos de dados. As organizações converteram muitas das suas interfaces telefônicas em interfaces da Web, e tornaram disponíveis on-line diversos serviços e informações. Por exemplo, ao acessar uma livraria on-line e procurar um livro ou coleção de música, você está acessando dados armazenados em um banco de dados. Quando você acessa um site de banco e recupera seu extrato e as informações da transação, estas são recuperadas do sistema de banco de dados do banco. Quando você acessa um site, as informações sobre você podem ser recuperadas de um banco de dados para selecionar que anúncios você deve ver. Além disso, os dados sobre seus acessos à Web podem ser armazenados em um banco de dados.

Portanto, embora as interfaces de usuário ocultem os detalhes do acesso a um banco de dados, e a maioria das pessoas nem mesmo tenha consciência de estar lidando com um banco de dados, acessar bancos de dados é uma parte essencial da vida de quase todo mundo hoje.

A importância dos sistemas de banco de dados pode ser julgada de outra maneira – atualmente, os fornecedores de sistemas de banco de dados, como Oracle, estão entre as maiores empresas de software do mundo, e os sistemas de banco de dados formam uma parcela importante da linha de produtos das empresas mais diversificadas, como Microsoft e IBM.

Finalidade dos sistemas de banco de dados

Os sistemas de banco de dados surgiram em resposta aos métodos mais antigos de gerenciamento computadorizado de dados comerciais. Como um exemplo desses métodos, típicos da década de 1960, considere parte de uma instituição bancária que, entre outros dados, mantém informações sobre todos os clientes e contas de poupança. Uma maneira de manter informações em um computador é armazená-las em arquivos de sistema operacional. Para permitir que os usuários manipulem as informações, o sistema possui diversos programas de aplicação que manipulam os arquivos, incluindo programas para:

- Debitar ou creditar uma conta
- Criar uma nova conta
- Fornecer o saldo de uma conta
- Gerar extratos mensais

Os programas de sistema escreveram esses programas de aplicação para atender às necessidades do banco.

Novos programas são acrescentados ao sistema conforme a necessidade. Por exemplo, suponha que um banco de poupança resolva oferecer contas-correntes. Como resultado, o banco cria novos arquivos permanentes que contêm informações sobre todas as contas-correntes mantidas no banco, e pode precisar escrever novos programas de aplicação para lidar com situações que não ocorrem nas contas de poupança, como, por exemplo, saldos negativos. Assim, com o passar do tempo, o sistema adquire mais arquivos e mais programas de aplicação.

Esse **sistema de processamento de arquivos** típico é suportado por um sistema operacional convencional. O sistema armazena registros permanentes em vários arquivos e precisa de diferentes programas de aplicação para extrair e acrescentar registros nos arquivos apropriados. Antes dos sistemas de gerenciamento de banco de dados (DBMSs), as organizações normalmente armazenavam as informações nesses sistemas.

Manter informações organizacionais em um sistema de processamento de arquivos apresenta diversas desvantagens importantes:

- **Redundância e inconsistência de dados.** Como diferentes programadores criam os arquivos e os programas de aplicação durante um longo período de tempo, os vários arquivos provavelmente precisam de diferentes estruturas, e os programas podem ser escritos em várias linguagens de programação. Além disso, as mesmas informações podem ser duplicadas em vários locais (arquivos). Por exemplo, o endereço e o número de telefone de um determinado cliente podem aparecer em um arquivo que consiste em registros de conta de poupança e em um arquivo que consiste em registros de conta-corrente. Essa redundância ocasiona um custo mais alto de armazenamento e acesso. Isso também pode causar uma **inconsistência de dados**; ou seja, as várias cópias dos mesmos dados podem não mais concordar. Por exemplo, um endereço de cliente alterado pode ser refletido em registros de conta de poupança, mas não em qualquer outro lugar do sistema.

- **Dificuldade de acesso a dados.** Suponha que um dos gerentes do banco precise encontrar os nomes de todos os clientes que moram em uma determinada área de CEP. O gerente pede essa lista ao departamento de processamento de dados. Como os projetistas do sistema original não previram esse tipo de solicitação, não há um programa de aplicação disponível para atendê-la. Há, no entanto, um programa de aplicação para gerar a lista de *todos* os clientes. O gerente agora tem duas escolhas: obter a lista de todos os clientes e extrair as informações necessárias manualmente ou pedir a um programador de sistemas para escrever o programa de aplicação necessário. Ambas as alternativas são obviamente insatisfatórias. Suponha que esse programa seja escrito e que, vários dias depois, o mesmo gerente precise reduzir essa lista para incluir apenas os clientes que têm um saldo bancário de $10.000 ou mais. Como você pode imaginar, não existe um programa para gerar essa lista. Novamente, o gerente tem as duas opções anteriores, nenhuma das quais é satisfatória.

  A questão aqui é que os ambientes de processamento de arquivo convencionais não permitem que os dados necessários sejam recuperados de uma maneira conveniente e eficaz. Sistemas de recuperação de dados mais receptivos são necessários para uso geral.
- **Isolamento de dados.** Como os dados estão dispersos em vários arquivos e os arquivos podem estar em diferentes formatos, escrever novos programas de aplicação para recuperar os dados apropriados se torna uma tarefa difícil.
- **Problemas de integridade.** Os valores de dados armazenados no banco de dados precisam satisfazer determinadas **restrições de consistência**. Por exemplo, o saldo de certos tipos de contas bancárias nunca pode ser inferior a uma quantia predeterminada (por exemplo, $25). Os desenvolvedores impõem essas restrições no sistema acrescentando código apropriado nos vários programas de aplicação. Entretanto, quando novas restrições são acrescentadas, é difícil mudar os programas para implementá-las. O problema é maior ainda quando as restrições envolvem vários itens de dados de arquivos diferentes.
- **Problemas de atomicidade.** Um sistema de computador, como qualquer outro dispositivo mecânico ou elétrico, está sujeito a falhas. Em muitas aplicações, é fundamental que, se ocorrer uma falha, os dados sejam restaurados ao estado consistente em que se encontravam antes da falha. Considere um programa para transferir $50 da conta *A* para a conta *B*. Se ocorrer uma falha durante a execução do programa, é possível que os $50 fossem removidos da conta *A* mas não fossem creditados na conta *B*, resultando em um estado de banco de dados inconsistente. Obviamente, é fundamental para a consistência do banco de dados que nem o débito nem o crédito ocorram. Ou seja, a transferência de fundos precisa ser *atômica* – ela precisa ocorrer em sua totalidade ou não deve ocorrer absolutamente. É difícil garantir atomicidade em um sistema de processamento de arquivo convencional.
- **Anomalias de acesso concorrente.** Para favorecer o desempenho geral do sistema e uma resposta mais rápida, muitos sistemas permitem que vários usuários atualizem os dados simultaneamente. Na verdade, hoje, os maiores revendedores da Internet podem ter milhões de acessos por dia aos seus dados pelos compradores. Nesse tipo de ambiente, a interação das atualizações concorrentes é possível e pode resultar em dados inconsistentes. Considere a conta bancária *A*, contendo $500. Se dois clientes retirarem fundos (digamos, $50 e $100, respectivamente) da conta *A* quase ao mesmo tempo, o resultado das execuções concorrentes pode deixar a conta em um estado incorreto (ou inconsistente). Suponha que os programas sendo executados em nome de cada retirada leiam o saldo anterior, diminuam esse valor pela quantidade sendo retirada e escrevam o resultado novamente. Se os dois programas forem executados concorrentemente, ambos podem ler o valor $500, e cada um pode escrever os resultados $450 e $400, respectivamente. Dependendo de qual escreva o valor por último, a conta pode conter $450 ou $400, em vez do valor correto, $350. Para se precaver dessa possibilidade, o sistema precisa manter algum tipo de supervisão. Isso, porém, é difícil de fornecer porque os dados podem ser acessados por muitos programas de aplicação diferentes que não foram previamente coordenados.
- **Problemas de segurança.** Nem todos os usuários do sistema de banco de dados devem ser capazes de acessar todos os dados. Por exemplo, em um sistema de banco, o pessoal da folha de pagamento precisa ver apenas a parte do banco de dados que contém as informações sobre os vários funcionários do banco. Eles não precisam acessar as informações sobre contas de clientes. No entanto, como os programas de aplicação são acrescentados ao sistema de processamento de arquivo de uma maneira provisória, é difícil impor essas restrições de segurança.

Essas dificuldades, entre outras, exigiram o desenvolvimento dos sistemas de banco de dados. Nas seções que se seguem, veremos os conceitos e os algoritmos que permitem que o sistema de banco de dados resolva os problemas com os sistemas de processamento de arquivo. Na maior parte deste livro, usamos uma instituição bancária como exemplo de uma aplicação de processamento de dados típica encontrada em uma organização.

Visão dos dados

Um sistema de banco de dados é uma coleção de dados inter-relacionados e um conjunto de programas que permitem aos usuários acessar e modificar esses dados. Uma importante finalidade de um sistema de banco de dados é fornecer aos usuários uma visão *abstrata* dos dados, ou seja, o sistema oculta certos detalhes de como os dados são armazenados e mantidos.

Abstração de dados

Para que o sistema seja funcional, ele precisa recuperar dados de maneira eficiente. A necessidade de eficiência tem levado projetistas a usar estruturas de dados complexas para representar dados no banco de dados. Como muitos usuários de sistema de banco de dados não são treinados em computador, os desenvolvedores ocultam a complexidade dos usuários sob vários níveis de abstração, para simplificar as interações do usuário com o sistema:

- **Nível físico.** O nível de abstração mais baixo descreve *como* os dados são realmente armazenados. O nível físico descreve em detalhes estruturas de dados complexas de baixo nível.
- **Nível lógico.** O próximo nível de abstração mais alto descreve *que* dados estão armazenados no banco de dados e que relações existem entre eles. O nível lógico, portanto, descreve o banco de dados inteiro em termos de um pequeno número de estruturas relativamente simples. Embora a implementação das estruturas simples no nível lógico possa envolver estruturas em nível físico complexas, o usuário do nível lógico não precisa estar consciente dessa complexidade. Os administradores de banco de dados, que precisam decidir que informações armazenar no banco de dados, usam o nível lógico de abstração.
- **Nível de view.** O nível de abstração mais alto descreve apenas parte do banco de dados. Mesmo que o nível lógico use estruturas mais simples, a complexidade permanece devido à variedade de informações armazenadas em um grande banco de dados. Muitos usuários do sistema de banco de dados não precisam de toda essa informação; em vez disso, eles precisam acessar apenas uma parte do banco de dados. O nível de view existe para simplificar sua interação com o sistema. O sistema pode fornecer muitas visões para o mesmo banco de dados.

A Figura 1.1 mostra a relação entre os três níveis de abstração.

Uma analogia para o conceito dos tipos de dados nas linguagens de programação pode esclarecer a diferença entre os níveis de abstração. A maioria das linguagens de programação aceita a noção de um tipo estruturado. Por exemplo, em uma linguagem do tipo Pascal, podemos declarar um registro da seguinte maneira:

```
type cliente = record
                 id_cliente : string;
                 nome_cliente : string;
                 rua_cliente : string;
                 cidade_cliente : string;
               end;
```

Esse código define um novo tipo de registro chamado *cliente* com quatro campos. Cada campo possui um nome e um tipo associado. Uma instituição bancária pode ter vários desses tipos de registro, incluindo

- *conta*, com campos *numero_conta* e *saldo*
- *funcionario*, com campos *nome_funcionario* e *salario*

No nível físico, um registro *cliente*, *conta* ou *funcionario* pode ser descrito como um bloco de locais de armazena-

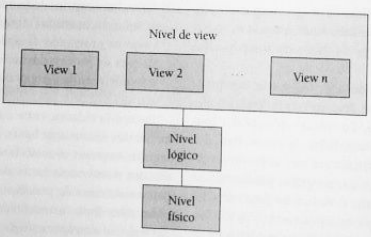


Figura 1.1 Os três níveis de abstração de dados.

mento consecutivos (por exemplo, words ou bytes). O compilador oculta esse nível de detalhe dos programadores. Da mesma forma, o sistema de banco de dados oculta muitos detalhes de armazenamento de nível mais baixo dos programadores de banco de dados. Os administradores de banco de dados, por outro lado, podem estar cientes de certos detalhes da organização física dos dados.

No nível lógico, cada registro é descrito por um tipo de informação, como no segmento de código anterior, e a relação desses tipos de registro também é definida. Os programadores que utilizam uma linguagem de programação trabalham nesse nível de abstração. Semelhantemente, os administradores de banco de dados normalmente trabalham nesse nível de abstração.

Finalmente, no nível de view, os usuários de computador vêem um conjunto de programas de aplicação que ocultam detalhes dos tipos de dados. De igual modo, no nível de view, várias visões do banco de dados são definidas e os usuários de banco de dados vêem essas visões. Além disso, para ocultar detalhes do nível lógico do banco de dados, as visões também fornecem um mecanismo de segurança de modo a evitar que os usuários acessem certas partes do banco de dados. Por exemplo, os caixas de um banco vêem apenas a parte do banco de dados que contém informações das contas de clientes; eles não podem acessar informações sobre salários dos funcionários.

Instâncias e esquemas

Os bancos de dados mudam com o tempo, à medida que as informações são inseridas e excluídas. A coleção das informações armazenadas no banco de dados em um determinado momento é uma **instância** do banco de dados. O projeto geral do banco de dados é o **esquema** do banco de dados. Os esquemas raramente – ou nunca – são modificados.

O conceito de esquemas e instâncias de banco de dados pode ser entendido por analogia com um programa escrito em uma linguagem de programação. Um esquema de banco de dados corresponde às declarações de variável (juntamente com as definições de tipo associadas) em um programa. Cada variável possui um valor específico em um dado instante. Os valores das variáveis em um programa em um ponto no tempo correspondem a uma *instância* de um esquema de banco de dados.

Os sistemas de banco de dados possuem vários esquemas, particionados de acordo com os níveis de abstração. O **esquema físico** descreve o projeto de banco de dados no nível físico, enquanto o **esquema lógico** descreve o projeto de banco de dados no nível lógico. Um banco de dados também pode ter vários esquemas no nível de view, algumas vezes chamado de **subesquemas**, que descrevem diferentes visões do banco de dados.

Desses, o esquema lógico é seguramente o mais importante, em termos de seu efeito sobre os programas de aplicação, já que os programadores constroem aplicações usando o esquema lógico. O esquema físico é oculto por trás do esquema lógico, e, em geral, pode ser facilmente modificado sem afetar os programas de aplicação. Dizemos que os programas de aplicação apresentam **independência de dados física** se eles não dependerem do esquema físico e, portanto, não precisarem ser reescritos se o esquema físico mudar.

Estudaremos as linguagens para descrever esquemas, após apresentarmos a noção dos modelos de dados na próxima seção.

Modelos de dados

Apoiando a estrutura de um banco de dados está o **modelo de dados**: uma coleção de ferramentas conceituais para descrever dados, relações de dados, semântica de dados e restrições de consistência. Um modelo de dados oferece uma maneira de descrever o projeto de um banco de dados no nível físico, lógico e de view.

Existem vários modelos de dados diferentes, que abordaremos neste livro. Os modelos de dados podem ser classificados em quatro categorias diferentes:

- **Modelo relacional.** O modelo relacional usa uma coleção de tabelas para representar os dados e as relações entre eles. Cada tabela possui diversas colunas, e cada coluna possui um nome único. O modelo relacional é um exemplo de um modelo baseado em registros. Os modelos baseados em registros recebem esse nome porque o banco de dados é estruturado em registros de formato fixo de vários tipos. Cada tabela contém registros de um tipo específico. Cada tipo de registro define um número fixo de campos, ou atributos. As colunas da tabela correspondem aos atributos do tipo de registro. O modelo de dados relacional é o modelo de dados mais usado, e uma grande maioria dos sistemas de banco de dados atuais é baseada no modelo relacional. Os Capítulos 2 a 7 abordam detalhadamente o modelo relacional.
- **Modelo de entidade/relacionamento.** O modelo de entidade/relacionamento (E-R) é baseado em uma percepção de um mundo real que consiste em uma coleção de objetos básicos, chamados *entidades*, e as *relações* entre esses objetos. Uma entidade é uma "coisa" ou "objeto" no mundo real que é distinguível dos outros objetos. O modelo de entidade/relacionamento é muito usado no projeto de banco de dados, e o Capítulo 6 o explora em profundidade.
- **Modelo de dados baseado em objeto.** O modelo de dados orientado a objeto é outro modelo que tem recebido

cada vez mais atenção. Ele pode ser visto como uma extensão ao modelo E-R com noções de encapsulamento, métodos (funções) e identidade de objeto. O modelo de dados relacional de objeto combina recursos do modelo de dados orientado a objeto e do modelo de dados relacional. O Capítulo 9 examina o modelo de dados baseado em objeto.

- **Modelo de dados semi-estruturado.** O modelo de dados semi-estruturado permite a especificação dos dados em que itens de dados individuais do mesmo tipo possam ter diferentes conjuntos de atributos. Isso é o oposto dos modelos de dados mencionados anteriormente, em que todos os itens de dados de um determinado tipo precisam ter o mesmo conjunto de atributos. A **Extensible Markup Language (XML)** é amplamente usada para representar dados semi-estruturados. O Capítulo 10 trata da XML.

Historicamente, o **modelo de dados de rede** e o **modelo de dados hierárquico** precederam o modelo de dados relacional. Esses modelos estavam intimamente relacionados com a implementação subjacente e complicavam a tarefa de modelar dados. Como resultado, eles são pouco usados atualmente, exceto em código de banco de dados antigo que ainda está em vigor em alguns locais. Eles são descritos resumidamente nos Apêndices A e B para os leitores interessados.

Linguagens de banco de dados

Um sistema de banco de dados fornece uma **linguagem de definição de dados** para especificar o esquema de banco de dados e uma **linguagem de manipulação de dados** para expressar as consultas e atualizações de banco de dados. Na prática, as linguagens de definição de dados e de manipulação de dados não são duas linguagens separadas, mas simplesmente formam partes de uma única linguagem de banco de dados, como a amplamente usada linguagem SQL.

Linguagem de manipulação de dados

Uma **linguagem de manipulação de dados (DML)** permite aos usuários acessar ou manipular dados conforme são organizados pelo modelo de dados apropriado. Os tipos de acesso são:

- Recuperação de informações armazenadas no banco de dados
- Inserção de novas informações no banco de dados
- Exclusão de informações do banco de dados
- Modificação de informações armazenadas no banco de dados

Basicamente, existem dois tipos:

- **DMLs procedurais** – Requerem que um usuário especifique *que* dados são necessários e *como* obtê-los.
- **DMLs declarativas** (também chamadas de **DMLs não procedurais**) – Requerem que um usuário especifique *que* dados são necessários *sem* especificar como obtê-los.

As DMLs declarativas normalmente são mais fáceis de aprender e de usar do que as DMLs procedurais. Entretanto, como um usuário não precisa especificar como obter os dados, o sistema de banco de dados precisa descobrir uma maneira eficiente de acessar os dados.

Uma **consulta** é uma instrução requisitando a recuperação de informações. A parte de uma DML que envolve recuperação de informações é chamada de **linguagem de consulta**. Embora tecnicamente incorreto, é uma prática comum usar os termos *linguagem de consulta* e *linguagem de manipulação de dados* sem distinção.

Existem várias linguagens de consulta de banco de dados em uso, quer comercial ou experimentalmente. Nos Capítulos 3 e 4, estudaremos a linguagem de consulta mais utilizada, a SQL. Também veremos algumas outras linguagens de consulta no Capítulo 5.

Os níveis de abstração que discutimos na seção "Visão dos dados" se aplicam não só para definir ou estruturar dados, mas também para manipular dados. No nível físico, precisamos definir algoritmos que permitam um acesso eficiente aos dados. Nos níveis mais altos de abstração, enfatizamos a facilidade de uso. O objetivo é permitir que os humanos interajam eficientemente com o sistema. O componente processador de consulta do sistema de banco de dados (que estudaremos nos Capítulos 13 e 14) traduz consultas de DML em sequências de ações no nível físico do sistema de banco de dados.

Linguagem de definição de dados

Especificamos um esquema de banco de dados por um conjunto de definições expresso por uma linguagem especial chamada **linguagem de definição de dados (DDL)**. A DDL também é usada para especificar propriedades adicionais dos dados.

Especificamos a estrutura de armazenamento e métodos de acesso usados pelo sistema de banco de dados por um conjunto de instruções em um tipo especial de DDL chamado de linguagem de **armazenamento e definição de dados**. Essas instruções definem os detalhes de implementação dos esquemas de banco de dados, que normalmente estão ocultos dos usuários.

Os valores de dados armazenados no banco de dados precisam satisfazer certas **restrições de consistência**. Por

exemplo, suponha que o saldo em uma conta não deva ser inferior a $100. A DDL oferece recursos para especificar essas restrições. Os sistemas de banco de dados verificam essas restrições toda vez que o banco de dados é atualizado. Em geral, uma restrição pode ser um predicado arbitrário pertencente ao banco de dados. Contudo, como predicados arbitrários podem ser difíceis de testar, os sistemas de banco de dados se preocupam com as restrições de integridade que podem ser testadas com overhead mínimo:

- **Restrições de domínio.** Um domínio dos possíveis valores precisa ser associado com cada atributo (por exemplo, tipos de inteiro, tipos de caractere, tipos de data/hora). Declarar um atributo para ser de um domínio específico age como uma restrição sobre os valores que ele pode assumir. As restrições de domínio são a forma mais elementar de restrição de integridade. Elas são testadas facilmente pelo sistema sempre que um novo item de dados é inserido no banco de dados.
- **Integridade referencial.** Existem casos em que desejamos garantir que um valor que aparece em uma relação para um determinado conjunto de atributos também apareça para um certo conjunto de atributos em outra relação (integridade referencial). As modificações de banco de dados podem causar violações da integridade referencial. Quando uma restrição de integridade referencial é violada, o procedimento normal é rejeitar a ação que causou a violação.
- **Assertivas.** Uma assertiva é qualquer condição que o banco de dados sempre precisa satisfazer. As restrições de domínio e as restrições de integridade referencial são formas especiais de assertivas. Todavia, existem muitas restrições que não podemos expressar usando apenas essas formas especiais. Por exemplo, "Todo empréstimo tem pelo menos um cliente que mantém uma conta com um saldo mínimo de $1.000" precisa ser expresso como uma assertiva. Quando uma assertiva é criada, o sistema testa sua validade. Se a assertiva for válida, qualquer futura modificação no banco de dados só será permitida se ela não fizer com que essa assertiva seja violada.
- **Autorização.** Podemos querer diferenciar entre usuários quanto ao tipo de acesso que eles podem realizar nos vários valores de dados no banco de dados. Essas diferenciações são expressas em termos de **autorização**, sendo que as mais comuns são: **autorização de leitura**, que permite a leitura, mas não a alteração, dos dados; **autorização de inserção**, que permite a inserção de novos dados, mas não a modificação de dados existentes; **autorização de atualização**, que permite a alteração, mas não a exclusão, dos dados; e **autorização de exclusão**, que permite a exclusão dos dados. Podemos atribuir o usuário a todas, nenhuma ou uma combinação desses tipos de autorização.

A DDL, exatamente como qualquer outra linguagem de programação, tem como entrada algumas instruções e gera alguma saída. A saída da DDL é colocada no **dicionário de dados**, que contém **metadados** – ou seja, dados sobre os dados. O dicionário de dados é considerado um tipo especial de tabela que só pode ser acessado e atualizado pelo próprio sistema de banco de dados (não por um usuário comum). Um sistema de banco de dados consulta o dicionário de dados antes de ler ou modificar dados reais.

Bancos de dados relacionais

Este tipo de banco de dados é baseado no modelo relacional e usa um conjunto de tabelas para representar os dados e as relações entre eles. Ele também inclui uma DML e uma DDL. A maioria dos sistemas de banco de dados relacionais utiliza a linguagem SQL, que discutiremos nesta seção e que abordaremos em mais detalhes nos Capítulos 3 e 4. No Capítulo 5, veremos outras importantes linguagens.

Tabelas

Cada tabela contém várias colunas, e cada coluna possui um nome único. A Figura 1.2 apresenta um banco de dados relacional de exemplo, constituído de três tabelas: a primeira mostra detalhes dos clientes do banco, a segunda mostra as contas e a terceira mostra que contas pertencem a que clientes.

A primeira tabela, *cliente*, mostra, por exemplo, que o cliente identificado pelo *id_cliente* 192-83-7465 é chamado Johnson e mora na rua Alma, nº 12, em Palo Alto. A segunda tabela, *conta*, mostra, por exemplo, que a conta A-101 possui um saldo de $500 e a conta A-201 possui um saldo de $900.

A terceira tabela mostra que contas pertencem a que clientes. Por exemplo, o número de conta A-101 pertence ao cliente cujo *id_cliente* é 192-83-7465, a saber, Johnson, e os clientes 192-83-7465 (Johnson) e 019-28-3746 (Smith) compartilham o número de conta A-201 (eles podem ser sócios de uma empresa comercial).

O modelo relacional é um exemplo de um modelo baseado em registro. Os modelos baseados em registro possuem esse nome porque o banco de dados é estruturado em registros de formato fixo de vários tipos. Cada tabela contém registros de um tipo específico. Cada tipo de registro define um número fixo de campos, ou atributos. As colunas da tabela correspondem aos atributos do tipo de registro.

Não é difícil ver como as tabelas podem ser armazenadas em arquivos. Por exemplo, um caractere especial (como uma vírgula) pode ser usado para delimitar diferentes atributos de um registro, e outro caractere especial (como um

id_cliente	*nome_cliente*	*rua_cliente*	*cidade_cliente*
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) Tabela *cliente*

número_conta	*saldo*
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) Tabela *conta*

id_cliente	*número_conta*
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) Tabela *depositante*

Figura 1.2 Exemplo de um banco de dados relacional.

caractere de nova linha) pode ser usado para delimitar registros. O modelo relacional oculta esses detalhes de implementação de baixo nível dos desenvolvedores e usuários de banco de dados.

O modelo de dados relacional é o mais utilizado e uma grande maioria dos sistemas de banco de dados atuais é baseada nele. Os Capítulos 2 a 7 abordam o modelo relacional em detalhes.

Também observamos que é possível criar esquemas no modelo relacional que têm problemas como informações desnecessariamente duplicadas. Por exemplo, suponha que armazenemos *número_conta* como um atributo do registro *cliente*. Então, para representar o fato de que as contas A-101 e A-201 pertencem ao cliente Johnson (com o *id_cliente* 192-83-7465), precisaríamos armazenar duas linhas na tabela *cliente*. Os valores para *nome_cliente*, *rua_cliente* e *cidade_cliente* para Johnson seriam desnecessariamente duplicados nas duas linhas. No Capítulo 7, estudaremos como distinguir bons e maus projetos de esquema.

Linguagem de manipulação de dados

A linguagem de consulta SQL é não procedural. Ela toma como entrada várias tabelas (possivelmente apenas uma) e sempre retorna uma única tabela. Aqui está um exemplo de uma consulta SQL que encontra nomes de todos os clientes que residem em Harrison:

```
select cliente.nome_cliente
from cliente
where cliente.cidade_cliente = 'Harrison'
```

A consulta especifica que as linhas da tabela *cliente* em que *cidade_cliente* é Harrison precisam ser recuperadas, e o atributo *nome_cliente* dessas linhas precisa ser exibido. Mais especificamente, o resultado de executar essa consulta é uma tabela com uma única coluna rotulada *nome_cliente*, e um conjunto de linhas, cada uma contendo o nome de um cliente cuja *cidade_cliente* é Harisson. Se a consulta for executada na tabela da Figura 1.2, o resultado consistirá em duas linhas, uma com o nome Hayes e a outra com o nome Jones.

As consultas podem envolver informações de mais de uma tabela. Por exemplo, a consulta a seguir localiza os números de conta e os saldos correspondentes de todas as contas pertencentes ao cliente com *id_cliente* 192-83-7465.

```
select conta.numero_conta, conta.saldo
from depositante, conta
where depositante.id_cliente = '192-83-7465' and
    depositante.numero_conta = conta.numero_conta
```

Se essa consulta fosse executada nas tabelas da Figura 1.2, o sistema descobriria que as duas contas de números A-101 e A-201 são pertencentes ao cliente 192-83-7465, e o resultado consistiria em uma tabela com duas colunas (*numero_conta, saldo*) e duas linhas (A-101, 500) e (A-201, 900).

Linguagem de definição de dados

A SQL fornece uma DDL rica que permite definir tabelas, restrições de integridade, assertivas etc.

Por exemplo, a instrução a seguir na linguagem SQL define a tabela *conta*:

```
create table conta
    (numero_conta char(10),
    saldo integer)
```

A execução dessa instrução DDL cria a tabela *conta*. Além disso, ela atualiza o dicionário de dados, que contém metadados (Linguagem de definição de dados). O esquema de uma tabela é um exemplo de metadados.

Acesso a bancos de dados por meio de programas de aplicação

A SQL não é tão poderosa quanto uma máquina universal de Turing; ou seja, existem algumas computações que não podem ser obtidas por qualquer consulta SQL. Essas computações precisam ser escritas em uma linguagem *host*, como Cobol, C, C++ ou Java, com consultas SQL embutidas que acessam os dados no banco de dados. **Programas de aplicação** são usados para interagir com o banco de dados dessa maneira. Exemplos em um sistema de banco são programas que geram contracheques, debitam contas, creditam contas ou transferem fundos entre contas.

Para acessar o banco de dados, as instruções de DML precisam ser executadas com a linguagem host. Existem duas maneiras de fazer isso:

- Fornecendo uma interface de programa de aplicação (conjunto de procedimentos) que pode ser usada para enviar instruções de DML e DDL para o banco de dados e recuperar os resultados.

  O padrão Open Database Connectivity (ODBC) definido pela Microsoft para uso com a linguagem C é um padrão de interface de programa de aplicação comumente usado. O padrão Java Database Connectivity (JDBC) fornece recursos correspondentes para a linguagem Java.
- Estendendo a sintaxe de linguagem host para incorporar chamadas de DML dentro do programa de linguagem host. Normalmente, um caractere especial antecede chamadas de DML, e um pré-processador, chamado **pré-compilador de DML**, converte as instruções de DML em chamadas de procedimento normais na linguagem host.

Projeto de um banco de dados

Os sistemas de banco de dados são projetados para gerenciar grandes blocos de informações. Esses grandes blocos de informações não existem isolados. Eles são parte da operação de alguma empresa cujo produto final pode ser informações do banco de dados ou pode ser algum dispositivo ou serviço para o qual o banco de dados desempenha apenas um papel de apoio.

O projeto de banco de dados envolve principalmente o projeto do esquema de banco de dados. O projeto de um ambiente de aplicação de banco de dados completo que atenda às necessidades da empresa sendo modelada requer atenção para um conjunto de problemas mais amplo. Neste texto, focalizamos inicialmente a escrita de consultas de banco de dados e o projeto de esquemas de banco de dados. O Capítulo 8 discute o processo geral do projeto de aplicação.

Processo do projeto

Um modelo de dados de alto nível serve ao projetista de banco de dados fornecendo uma estrutura conceitual para especificar, de uma maneira sistemática, quais são as necessida-

des de dados dos usuários do banco de dados, e como o banco de dados será estruturado para satisfazer essas necessidades. Portanto, a fase inicial do projeto de banco de dados é caracterizar completamente as necessidades de dados dos usuários de banco de dados potenciais. O projetista de banco de dados precisa interagir extensivamente com especialistas e usuários do domínio para realizar essa tarefa. O resultado dessa fase é uma especificação das necessidades do usuário.

A seguir, o projetista escolhe um modelo de dados e, aplicando os conceitos do modelo de dados escolhido, traduz essas necessidades em um esquema conceitual do banco de dados. O esquema desenvolvido nessa fase de **projeto conceitual** fornece uma visão geral detalhada da empresa. O projetista revisa o esquema para confirmar se todas as necessidades de dados estão realmente satisfeitas e se não estão em conflito entre si. O projetista também pode examinar o projeto para remover quaisquer recursos redundantes. O foco nesse momento é descrever os dados e suas relações, e não especificar detalhes do armazenamento físico.

Em termos do modelo relacional, o processo do projeto conceitual envolve decisões sobre *que* atributos queremos capturar no banco de dados e *como agrupar* esses atributos para formar as várias tabelas. A parte "que" é basicamente uma decisão empresarial e não a discutiremos mais neste texto. A parte "como" é principalmente um problema de ciência da computação. Existem basicamente duas maneiras de tratar o problema. A primeira é usar o modelo de entidade/relacionamento (seção "O modelo de entidade/relacionamento"); a outra é empregar um conjunto de algoritmos (coletivamente chamados de *normalização*) que tem como entrada o conjunto de todos os atributos e gera um conjunto de tabelas (seção "Normalização").

Um esquema conceitual totalmente desenvolvido também indicará as necessidades funcionais da empresa. Em uma **especificação das necessidades funcionais**, os usuários descrevem os tipos de operações (ou transações) que serão realizadas nos dados. Exemplos de operações incluem modificar ou atualizar dados, pesquisar e recuperar dados específicos e excluir dados. Nessa fase do projeto conceitual, o projetista pode revisar o esquema para se certificar de que ele atende as necessidades funcionais.

O processo de mudar de um modelo de dados abstrato para a implementação do banco de dados é realizado em duas fases de projeto finais. Na **fase de projeto lógico**, o projetista mapeia o esquema conceitual de alto nível para o modelo de dados de implementação do sistema de banco de dados que será usado. O projetista usa o esquema de banco de dados específico do sistema resultante na **fase de projeto físico** subsequente, em que os recursos físicos do banco de dados são especificados. Esses recursos, que incluem a forma de organização de arquivo e as estruturas de armazenamento internas, são discutidos no Capítulo 11.

Projeto de banco de dados para instituição bancária

Para ilustrar o processo do projeto, vamos examinar como um banco de dados para uma instituição bancária poderia ser projetado. A especificação inicial das necessidades de usuário pode ser baseada em entrevistas com os usuários do banco de dados e na análise da empresa pelo próprio projetista. A descrição que surge dessa fase de projeto serve como a base para especificar a estrutura conceitual do banco de dados. Aqui estão as principais características da instituição bancária.

- O banco é organizado em agências. Cada agência está localizada em uma determinada cidade e é identificada por um nome único. O banco controla os ativos de cada agência.
- Os clientes do banco são identificados por seus valores de *id_cliente*. O banco armazena o nome de cada cliente e a rua e cidade onde o cliente reside. Os clientes podem ter contas e podem fazer empréstimos. Um cliente pode estar associado a um banqueiro específico, que pode agir como um gerente de empréstimo ou como um banqueiro pessoal para esse cliente.
- O banco oferece dois tipos de conta – poupança e conta-corrente. As contas podem ser mantidas por mais de um cliente, e um cliente pode ter mais de uma conta. A cada conta é atribuído um número de conta único. O banco mantém um registro do saldo de cada conta e a data mais recente em que a conta foi acessada por cliente titular da conta. Além disso, cada conta de poupança tem uma taxa de juros e os saques a descoberto são registrados para cada conta-corrente.
- O banco fornece empréstimos a seus clientes. Um empréstimo se origina em uma agência específica e pode ser mantido por um ou mais clientes. Um empréstimo é identificado por um número de empréstimo único. Para cada empréstimo, o banco controla a quantidade e os pagamentos do empréstimo. Embora um número de pagamento de empréstimo não identifique unicamente um determinado pagamento entre aqueles para todos os empréstimos do banco, um número de pagamento identifica um pagamento específico para um empréstimo específico. A data e a quantia são registradas para cada pagamento.
- Os funcionários do banco são identificados por seus valores de *id_funcionário*. A administração do banco armazena o nome e o número de telefone de cada funcionário, os nomes dos dependentes do funcionário e o número *id_funcionário* do gerente do funcionário. O banco também controla a data de admissão do funcionário e, portanto, a duração do emprego.

Em uma instituição bancária real, o banco monitoraria os depósitos e saques das contas de poupança e corrente, exatamente como monitora os pagamentos para contas de empréstimo. Como as necessidades de modelagem para esse controle são semelhantes e gostaríamos de manter nossa aplicação de exemplo pequena, não monitoramos esses depósitos e saques em nosso modelo.

O modelo de entidade/relacionamento

O modelo de dados de entidade/relacionamento (E-R) é baseado em uma percepção de um mundo real que consiste em uma coleção de objetos básicos, chamados *entidades*, e de *relações* entre esses objetos. Uma entidade é uma "coisa" ou "objeto" no mundo real que é distinguível de outros objetos. Por exemplo, cada pessoa é uma entidade, e as contas bancárias podem ser consideradas entidades.

As entidades são descritas em um banco de dados por um conjunto de **atributos**. Por exemplo, os atributos *número_conta* e *saldo* podem descrever uma conta específica em um banco e formam atributos do conjunto de entidade *conta*. Da mesma maneira, os atributos *nome_cliente*, *rua_cliente* e *cidade_cliente* podem descrever uma entidade *cliente*.

Um atributo extra, *id_cliente*, é usado para identificar clientes unicamente (já que é possível ter dois clientes com o mesmo nome, rua e cidade). Um identificador de cliente único precisa ser atribuído a cada cliente. Nos Estados Unidos, muitas empresas usam o número do seguro social de uma pessoa (um número único que o governo atribui a cada pessoa no país) como um identificador de cliente.

Uma **relação** é uma associação entre várias entidades. Por exemplo, uma relação *depositante* associa um cliente a cada conta que ele possui. O conjunto de todas as entidades do mesmo tipo e o conjunto de todas as relações do mesmo tipo são chamados, respectivamente, de **conjunto de entidade** e **conjunto de relação**.

A estrutura lógica geral (esquema) de um banco de dados pode ser expressa graficamente por um *diagrama E-R*, constituído dos seguintes componentes:

- **Retângulos**, que representam conjuntos de entidade
- **Elipses**, que representam atributos
- **Losangos**, que representam conjuntos de relações entre um membro de cada um dos vários conjuntos de entidade
- **Linhas**, que ligam atributos a conjuntos de entidade e conjuntos de entidade a relações.

Cada componente é rotulado com a entidade ou relação que ele representa.

Como ilustração, considere parte de um sistema de banco de dados bancário consistindo em clientes e nas contas que esses clientes possuem. A Figura 1.3 mostra o diagrama E-R correspondente. O diagrama indica que existem dois conjuntos de entidade, *cliente* e *conta*, com atributos como descrito anteriormente. O diagrama também mostra uma relação *depositante* entre cliente e conta.

Além das entidades e relações, o modelo E-R representa certas restrições às quais o conteúdo de um banco de dados precisa se conformar. Uma restrição importante é a **cardinalidade de mapeamento**, que expressa o número de entidades ao qual outra entidade pode estar associada por meio de um conjunto de relação. Por exemplo, se cada conta precisa pertencer a apenas um cliente, o modelo E-R pode expressar essa restrição.

O modelo de entidade/relacionamento é amplamente usado no projeto de banco de dados, e o Capítulo 6 o analisa em detalhes.

Normalização

Outro método para projetar um banco de dados relacional é usar um processo normalmente conhecido como normalização. O objetivo é gerar um conjunto de esquemas de relação que permita armazenar informações sem redundância desnecessária, ao mesmo tempo permitindo recuperar informações facilmente. A técnica é projetar esquemas que estejam em uma *forma normal* apropriada. Para determinar se um esquema de relação é uma das formas normais desejáveis, precisamos de informações adicionais sobre a em-

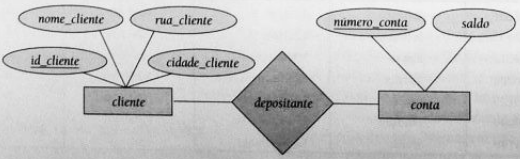


Figura 1.3 Exemplo de um diagrama E-R.

id_cliente	*numero_conta*	*saldo*
192-83-7465	A-101	500
192-83-7465	A-201	900
019-28-3746	A-215	700
677-89-9011	A-102	400
182-73-6091	A-305	350
321-12-3123	A-217	750
336-66-9999	A-222	700
019-28-3746	A-201	900

Figura 1.4 Tabela *depositante*.

presa real que estamos modelando com o banco de dados. O método mais comum é usar **dependências funcionais**, que discutimos na seção "Teoria da dependência funcional" do Capítulo 7.

Para entender a necessidade da normalização, vejamos o que pode acontecer de errado em um projeto de banco de dados. Entre as propriedades indesejáveis que um projeto ruim pode ter estão:

- Repetição de informações
- Incapacidade de representar certas informações

Devemos discutir esses problemas com a ajuda de um projeto de banco de dados modificado para nosso exemplo de banco.

Suponha que, em vez de ter as duas tabelas separadas *conta* e *depositante*, tenhamos uma única tabela, *depositante'*, que combina as informações das duas tabelas (como ilustrado na Figura 1.4). Observe que existem duas linhas em *depositante'* que contêm informações sobre a conta A-201. Desejamos evitar a repetição das informações em nosso projeto alternativo porque isso ocupa espaço, além de complicar a atualização do banco de dados. Suponha que desejamos mudar o saldo da conta A-201 de $900 para $950. Essa alteração precisa se refletir nas duas linhas; compare-a com o projeto original, em que isso resultará na atualização em uma única linha. Portanto, as atualizações são mais custosas no projeto alternativo do que no projeto original. Quando realizamos a atualização no banco de dados alternativo, precisamos garantir que *cada* tupla pertencente à conta A-201 seja atualizada, ou nosso banco de dados mostrará dois valores de saldo diferentes para a conta A-201.

Vamos voltar nossa atenção para o problema da "incapacidade de representar certas informações". Suponha que, em vez de ter as duas tabelas separadas *cliente* e *depositante*, tenhamos uma única tabela, *cliente'*, que combina as informações das duas tabelas (como ilustrado na Figura 1.5). Não podemos representar diretamente as informações relativas a um cliente (*id_cliente*, *nome_cliente*, *rua_cliente*, *cidade_cliente*) a menos que ele tenha pelo menos uma conta no banco. Isso é porque as linhas na tabela *cliente'* exigem valores para *numero_conta*.

Uma solução para esse problema é introduzir valores **nulos**. O valor *nulo* indica que esse valor não existe (ou não é conhecido). Um valor não conhecido pode ser *omisso* (o valor existe, mas não temos essa informação) ou *desconhecido* (não sabemos se o valor realmente existe ou não). Como veremos mais adiante, os valores nulos são difíceis

id_cliente	*nome_cliente*	*rua_cliente*	*cidade_cliente*	*numero_conta*
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-101
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-201
677-89-9011	Hayes	3 Main St.	Harrison	A-102
182-73-6091	Turner	123 Putnam Ave.	Stamford	A-305
321-12-3123	Jones	100 Main St.	Harrison	A-217
336-66-9999	Lindsay	175 Park Ave.	Pittsfield	A-222
019-28-3746	Smith	72 North St.	Rye	A-201

Figura 1.5 Tabela *cliente*.

de manipular, e é preferível não lançar mão deles. Se não estivermos dispostos a lidar com valores nulos, podemos criar um determinado item de informação do cliente apenas quando ele tiver uma conta no banco (note que um cliente pode ter um empréstimo, mas não uma conta). Além disso, precisamos excluir essas informações quando o cliente fechar sua conta. Obviamente, a situação é indesejável, já que, em nosso projeto de banco de dados original, as informações do cliente estariam disponíveis quer ele tivesse ou não uma conta no banco, e sem utilizar valores nulos.

Bancos de dados semi-estruturados e baseados em objeto

Várias áreas de aplicação para os sistemas de banco de dados são limitadas pelas restrições do modelo de dados relacional. Como resultado, os pesquisadores desenvolveram vários modelos de dados para lidar com esses domínios de aplicação. Os modelos de dados que abordaremos no texto são os modelos de dados orientados a objeto e relacional de objeto (que representam os modelos de dados baseados em objeto) e a XML (que representa os modelos de dados semi-estruturados).

Modelos de dados baseados em objeto

O **modelo de dados orientado a objeto** é baseado no paradigma de linguagem de programação orientada a objeto, muito utilizada atualmente. Herança, identidade de objeto e encapsulamento (ocultação de informações), com métodos para fornecer uma interface para objetos, estão entre os principais conceitos da programação orientada a objeto que encontraram usos na modelagem de dados. O modelo de dados orientado a objeto também aceita um sistema de tipo rico, incluindo tipos estruturados e de coleção. O modelo orientado a objeto pode ser visto como uma extensão do modelo E-R com noções de encapsulamento, métodos (funções) e identidade de objeto.

O **modelo de dados relacional de objeto** estende o modelo relacional tradicional com uma variedade de recursos, como tipos estruturados e de coleção, bem como orientação a objeto.

O Capítulo 9 examina os bancos de dados relacionais de objeto (ou seja, os bancos de dados construídos sob o modelo relacional de objeto), assim como os bancos de dados orientados a objeto (ou seja, bancos de dados construídos sob o modelo de dados orientado a objeto).

Modelos de dados semi-estruturados

Os modelos de dados semi-estruturados permitem a especificação dos dados, em que itens de dados individuais do mesmo tipo podem ter diferentes conjuntos de atributos. Isso está em contraste com os modelos de dados mencionados anteriormente, em que cada item de dados de um tipo específico precisa ter o mesmo conjunto de atributos.

A linguagem XML foi inicialmente projetada como uma maneira de incluir informações de marcação em documentos de texto, mas se tornou importante devido a suas aplicações na troca de dados. A XML oferece um meio de representar dados que tenham estrutura aninhada e, além disso, permite uma grande flexibilidade na estruturação dos dados, o que é importante para certos tipos de dados não tradicionais. O Capítulo 10 descreve a linguagem XML, além de diferentes maneiras de expressar consultas nos dados representados em XML e de transformar dados XML de uma forma em outra.

Armazenamento e consulta de dados

Um sistema de banco de dados é particionado em módulos que lidam com cada uma das responsabilidades do sistema geral. Os componentes funcionais de um sistema de banco de dados podem ser divididos, de modo amplo, nos componentes gerenciador de armazenamento e processador de consulta.

O gerenciador de armazenamento é importante porque os bancos de dados normalmente exigem uma grande quantidade de espaço de armazenamento. Os bancos de dados corporativos variam de centenas de gigabytes a – para os maiores bancos de dados – terabytes de dados. Um gigabyte possui 1.000 megabytes (ou 1 bilhão de bytes) e um terabyte possui 1 milhão de megabytes (ou 1 trilhão de bytes). Como a memória principal dos computadores não pode armazenar tanta quantidade de dados, as informações são armazenadas em discos. Os dados são movidos entre o armazenamento em disco e a memória principal conforme necessário. Uma vez que o movimento de dados de e para o disco é lento, em comparação à velocidade da CPU, é fundamental que o sistema de banco de dados estruture os dados de modo a minimizar a necessidade de mover dados entre disco e memória principal.

O processador de consulta é importante porque ajuda o sistema de banco de dados a simplificar o acesso aos dados. As visões de alto nível ajudam a alcançar esse objetivo; com elas, os usuários do sistema não são desnecessariamente afligidos com os detalhes físicos da implementação do sistema. Entretanto, o rápido processamento das atualizações e consultas é importante. É função do sistema de banco de dados traduzir atualizações e consultas escritas em uma linguagem não procedural, no nível lógico, em uma seqüência eficiente de operações no nível físico.

Gerenciador de armazenamento

Um *gerenciador de armazenamento* é um módulo de programa que fornece a interface entre os dados de baixo nível armazenados no banco de dados e os programas de aplicação e consultas submetidos ao sistema. O gerenciador de armazenamento é responsável pela interação com o gerenciador de arquivos. Os dados brutos são armazenados no disco usando o sistema de arquivos, que normalmente é fornecido por um sistema operacional convencional. O gerenciador de armazenamento traduz as várias instruções DML em comandos de sistema de arquivos de baixo nível. Portanto, o gerenciador de armazenamento é responsável por armazenar, recuperar e atualizar dados no banco de dados.

Os componentes do gerenciador de armazenamento incluem:

- **Gerenciador de autorização e integridade**, que testa a satisfação das restrições de integridade e verifica a autoridade dos usuários para acessar dados.
- **Gerenciador de transação**, que garante que o banco de dados permaneça em um estado consistente (correto), apesar de falhas do sistema, e que as execuções de transação concorrentes sejam efetuadas sem conflito.
- **Gerenciador de arquivos**, que controla a alocação de espaço no armazenamento de disco e as estruturas de dados usadas para representar informações armazenadas no disco.
- **Gerenciador de buffer**, que é responsável por buscar dados do armazenamento de disco para a memória principal e decidir que dados colocar em cache na memória principal. O gerenciador de buffer é uma parte crítica do sistema de banco de dados, já que permite que o banco de dados manipule tamanhos de dados que sejam muito maiores do que o tamanho da memória principal.

O gerenciador de armazenamento implementa várias estruturas de dados como parte da implementação do sistema físico:

- **Arquivos de dados**, que armazenam o banco de dados propriamente dito.
- **Dicionário de dados**, que armazena metadados sobre a estrutura do banco de dados, em especial o esquema do banco de dados.
- **Índices**, que podem fornecer acesso rápido aos itens de dados. Como o índice deste livro, um índice de banco de dados fornece ponteiros para os itens de dados que contêm um valor específico. Por exemplo, poderíamos usar um índice para encontrar todos os registros de *conta* com um determinado *número_conta*. O hashing é uma alternativa para a indexação que é mais rápida em alguns casos, mas não em todos eles.

Discutiremos o meio de armazenamento, as estruturas de arquivo e o gerenciador de buffer no Capítulo 11. Os métodos de acessar dados eficientemente por meio de indexação ou hashing são discutidos no Capítulo 12.

O processador de consulta

Os componentes do processador de consulta incluem

- **Interpretador de DDL**, que interpreta instruções DDL e registra as definições no dicionário de dados.
- **Compilador de DML**, que traduz instruções DML em uma linguagem de consulta para um plano de avaliação consistindo em instruções de baixo nível que o mecanismo de avaliação de consulta entende.

  Uma consulta normalmente pode ser traduzida em qualquer um de vários planos de avaliação que produzem todos o mesmo resultado. O compilador de DML também realiza **otimização de consulta**; ou seja, ele seleciona o plano de avaliação de menor custo dentre as alternativas.
- **Mecanismo de avaliação de consulta**, que executa instruções de baixo nível geradas pelo compilador de DML.

A avaliação de consulta é abordada no Capítulo 13, enquanto os métodos pelos quais o otimizador de consulta escolhe dentre as possíveis estratégias de avaliação são discutidos no Capítulo 14.

Gerenciamento de transação

Muitas vezes, várias operações no banco de dados formam uma única unidade lógica de trabalho. Um exemplo é uma transferência de fundos, como na seção "Finalidade dos sistemas de banco de dados", no início deste capítulo, em que uma conta (digamos, A) é debitada e outra conta (digamos, B) é creditada. Obviamente, é fundamental que tanto o débito quanto o crédito ocorram, ou então que nenhum ocorra. Ou seja, a transferência de fundos precisa acontecer em sua totalidade ou não acontecer. Essa propriedade "tudo ou nada" é chamada **atomicidade**. Além disso, é essencial que a execução da transferência de fundos preserve a consistência do banco de dados. Isto é, o valor da soma $A + B$ precisa ser preservado. Essa propriedade de exatidão é chamada **consistência**. Finalmente, após a execução bem-sucedida de uma transferência de fundos, os novos valores das contas A e B precisa persistir, apesar da possibilidade de falha do sistema. Essa propriedade de persistência é chamada **durabilidade**.

Uma **transação** é um conjunto de operações que realiza uma única função lógica em uma aplicação de banco de dados. Cada transação é uma unidade da atomicidade e da

consistência. Assim, exigimos que as transações não violem quaisquer restrições de consistência de banco de dados. Ou seja, se o banco de dados tiver sido consistente quando uma transação tiver iniciado, o banco de dados precisa ser consistente quando a transação terminar com sucesso. Todavia, durante a execução de uma transação, pode ser necessário permitir temporariamente a inconsistência, já que o débito de *A* ou o crédito de *B* precisa ser feito antes do outro. Essa inconsistência temporária, embora necessária, pode causar dificuldade se uma falha ocorrer.

É responsabilidade do programador definir corretamente as várias transações, de modo que cada uma preserve a consistência do banco de dados. Por exemplo, a transação para transferir fundos da conta *A* para a conta *B* poderia ser definida para ser composta de dois programas separados: um que débita a conta *A* e outro que credita a conta *B*. A execução desses dois programas em seqüência sem dúvida preservará a consistência. Entretanto, cada programa, sozinho, não transforma o banco de dados de um estado consistente em um novo estado consistente. Portanto, esses programas não são transações.

Garantir as propriedades da atomicidade e da durabilidade é função do próprio sistema de banco de dados – especificamente, do **componente de gerenciamento de transação**. Na ausência de falhas, todas as transações completam com sucesso, e a atomicidade é conseguida facilmente. Contudo, devido aos vários tipos de falha, uma transação nem sempre pode completar sua execução corretamente. Se precisamos garantir a propriedade da atomicidade, uma transação falha não pode ter qualquer efeito sobre o estado do banco de dados. Portanto, o banco de dados precisa ser restaurado ao estado em que estava antes de a transação em questão ser iniciada. O sistema de banco de dados precisa realizar **recuperação de falha**, ou seja, detectar falhas de sistema e restaurar o banco de dados ao estado anterior à falha.

Finalmente, quando várias transações atualizam o banco de dados ao mesmo tempo, a consistência dos dados pode não mais ser preservada, mesmo que cada transação individual esteja correta. É responsabilidade do **gerenciador de controle de concorrência** controlar a interação entre as transações concorrentes, para assegurar a consistência do banco de dados.

Os conceitos básicos do processamento de transação são discutidos no Capítulo 15. O gerenciamento das transações concorrentes é abordado no Capítulo 16. O Capítulo 17 analisa detalhadamente a recuperação de falhas.

Os sistemas de banco de dados projetados para uso em pequenos computadores pessoais podem não ter todos esses recursos. Por exemplo, muitos sistemas pequenos permitem que apenas um usuário acesse o banco de dados de cada vez. Outros não oferecem backup ou recuperação, deixando essas funções para o usuário. Essas restrições permitem um gerenciamento de dados menor, com menos necessidades de recursos físicos – especialmente memória principal. Embora esse método de baixo custo e baixa capacidade seja adequado para pequenos bancos de dados pessoais, ele é inadequado para uma empresa de média ou grande escala.

O conceito de uma transação foi aplicado amplamente nos sistemas e aplicações de banco de dados. Embora o uso inicial das transações tenha sido em aplicações financeiras, o conceito agora é usado em aplicações em tempo real na telecomunicação, bem como no gerenciamento de atividades de longa duração, como projeto de produto ou workflows administrativos. Essas aplicações mais amplas do conceito de transação são discutidas no Capítulo 25.

Análise e mineração de dados

O termo mineração **de dados** (Data Mining) se refere aproximadamente ao processo de analisar de forma semi-automática grandes bancos de dados para encontrar padrões úteis. Como a descoberta de conhecimento na inteligência artificial (também chamada **aprendizado de máquina**) ou análise estatística, a mineração de dados usa os dados para descobrir regras e padrões. Entretanto, a mineração de dados difere do aprendizado de máquina e da estatística à medida que lida com grandes volumes de dados, armazenados principalmente no disco. Ou seja, a mineração de dados lida com "descoberta de conhecimento em bancos de dados".

Alguns tipos de conhecimento descobertos a partir de um banco de dados podem ser representados por um conjunto de **regras**. Este é um exemplo de uma regra, enunciada informalmente: "Mulheres jovens com renda anual maior que $50.000 são as pessoas mais propensas a comprar pequenos carros esportivos". É claro, essas regras não são universalmente verdadeiras, mas possuem graus de "suporte" e "confiança". Outros tipos de conhecimento são representados por equações relacionando diferentes variáveis entre si, ou por outros mecanismos para prever resultados quando os valores de algumas variáveis são conhecidos.

Existem vários tipos possíveis de padrão que podem ser úteis, e diferentes técnicas são usadas para encontrar diferentes tipos de padrão. No Capítulo 18 estudaremos alguns exemplos de padrões e veremos como eles podem ser automaticamente derivados de um banco de dados.

Normalmente, existe um componente manual para a mineração de dados, consistindo em pré-processar dados em uma forma aceitável para os algoritmos e pós-processar padrões descobertos para encontrar novos padrões que possam ser úteis. Também pode existir mais de um tipo de padrão ser descoberto de um determinado banco de dados,

e pode ser necessária interação manual para escolher tipos úteis de padrões. Por essa razão, a mineração de dados é, na verdade, um processo semi-automático na vida real. Todavia, em nossa descrição, nos concentraremos no aspecto automático da mineração.

As empresas começaram a explorar os dados proliferados on-line para tomar decisões mais acertadas sobre suas atividades, tal como que itens estocar e como atingir melhor os clientes para aumentar as vendas. No entanto, muitas de suas consultas são bastante complicadas, e certos tipos de informações não podem ser extraídos mesmo usando SQL.

Várias técnicas e ferramentas estão disponíveis para ajudar na decisão. Várias ferramentas para análise de dados permitem que os analistas vejam os dados de diferentes maneiras. Outras ferramentas de análise pré-calculam resumos de quantidades extremamente grandes de dados, a fim de fornecer respostas rápidas a consultas. Agora, o padrão SQL:1999 contém construções adicionais para aceitar análise de dados.

Os dados textuais também tiveram um crescimento explosivo. Eles são desestruturados, ao contrário dos dados rigidamente estruturados nos bancos de dados relacionais. A consulta de dados textuais desestruturados é chamada de *recuperação de informações*. Os sistemas de recuperação de informações têm muito em comum com os sistemas de banco de dados – em especial, o armazenamento e a recuperação de dados no armazenamento secundário. Entretanto, a ênfase no campo dos sistemas de informação é diferente da ênfase nos sistemas de banco de dados. Ela se concentra em aspectos como consulta baseada em palavras-chave, a relevância de documentos para a consulta e a análise, classificação e indexação de documentos. Nos Capítulos 18 e 19, abordamos o suporte à decisão, incluindo processamento analítico on-line, mineração de dados e recuperação de informações.

Arquitetura do banco de dados

Estamos agora em posição para fornecer um único quadro (Figura 1.6) dos vários componentes de um sistema de banco de dados e as conexões entre eles.

A arquitetura de um sistema de banco de dados é bastante influenciada pelo sistema de computador subjacente em que o sistema de banco de dados é executado. Os sistemas de banco de dados podem ser centralizados, ou cliente-servidor, com uma máquina servidora executando trabalho em nome de várias máquinas clientes. Os sistemas de banco de dados também podem ser projetados para explorar arquiteturas de computador paralelas. Os bancos de dados distribuídos abrangem múltiplas máquinas geograficamente separadas.

No Capítulo 20 abordamos a estrutura geral dos modernos sistemas de computador. O Capítulo 21 descreve como as várias ações de um banco de dados, sobretudo o processamento de consulta, podem ser implementadas para explorar o processamento paralelo. O Capítulo 22 apresenta diversos problemas que surgem em um banco de dados distribuído, e descreve como lidar com cada problema. Os problemas incluem como armazenar dados, como garantir a atomicidade das transações que são executadas em vários locais, como realizar controle de concorrência e como fornecer alta disponibilidade na presença de falhas. O processamento de consulta distribuído e os sistemas de diretório também são descritos nesse capítulo.

A maioria dos usuários de um sistema de banco de dados atualmente não está presente no local do sistema de banco de dados, mas, sim, conectada a ele através de uma rede. Portanto, podemos diferenciar entre máquinas **clientes**, em que usuários de banco de dados remotos trabalham, e máquinas **servidoras**, em que o sistema de banco de dados é executado.

As aplicações de banco de dados normalmente são particionadas em duas ou três partes, como na Figura 1.7. Em uma **arquitetura de duas camadas**, a aplicação é particionada em um componente que reside na máquina cliente, que chama a funcionalidade do sistema de banco de dados na máquina servidora por meio de instruções da linguagem. Os padrões de interface de programa de aplicação, como ODBC e JDBC, são usados para interação entre o cliente e o servidor.

Já em uma **arquitetura de três camadas**, a máquina cliente age meramente como um front-end e não contém quaisquer chamadas de banco de dados diretas. Em vez disso, o cliente finaliza a comunicação com um **servidor de aplicação**, normalmente por meio de uma interface de formulários. O servidor de aplicação, por sua vez, se comunica com um sistema de banco de dados para acessar os dados. A **lógica empresarial** da aplicação, que diz quais ações executar sob quais condições, é incorporada ao servidor de aplicação, em vez de ser distribuída por múltiplos clientes. As aplicações de três camadas são mais apropriadas para grandes aplicações e para aquelas executadas na World Wide Web.

Usuários e administradores de banco de dados

Um importante objetivo de um sistema de banco de dados é recuperar informações do banco de dados e armazenar novas informações nele. As pessoas que trabalham com um banco de dados podem ser categorizadas como usuários de banco de dados ou **administradores** de banco de dados.

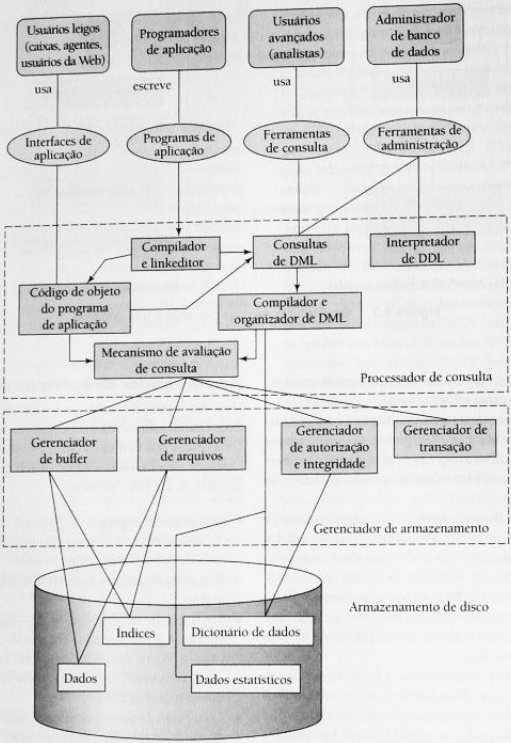


Figura 1.6 Estrutura do sistema.

Usuários de banco de dados e interfaces com o usuário

Existem quatro tipos diferentes de usuários de sistema de banco de dados, distinguidos pela maneira como esperam interagir com o sistema. Diversos tipos de interfaces com o usuário foram projetados para os diferentes tipos de usuários.

- **Usuários leigos** – São usuários não avançados que interagem com o sistema chamando um dos programas de aplicação previamente escritos. Por exemplo, um caixa bancário que precisa transferir $50 da conta *A* para a conta *B* chama um programa denominado *transferência*. Esse programa solicita ao caixa a quantia a ser transferida, a conta da qual o dinheiro deve ser transferido e a conta para a qual o dinheiro será transferido.

  Como outro exemplo, considere um usuário que deseja consultar seu saldo bancário na Web. Ele pode acessar um formulário, no qual insere o número de sua conta. Um programa de aplicação no servidor Web, então,

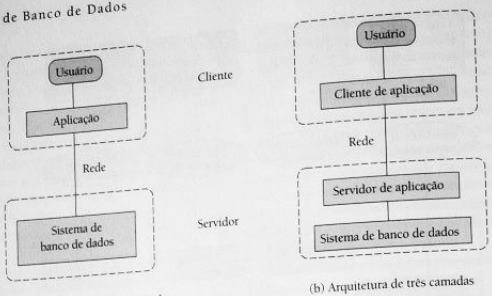


Figura 1.7 Arquiteturas de duas camadas e três camadas.

recupera o saldo da conta, usando o número de conta informado, e passa essa informação para o usuário.

A interface típica para usuários leigos é composta de formulários, na qual o usuário pode preencher campos apropriados no formulário. Os usuários leigos também podem simplesmente ler *relatórios* gerados do banco de dados.

- **Programadores de aplicação** – São profissionais de computação que escrevem programas de aplicação. Os programadores de aplicação podem escolher entre muitas ferramentas para desenvolver interfaces com o usuário. As ferramentas de RAD (**Rapid Application Development**) permitem que um programador de aplicação construa formulários e relatórios com um mínimo de esforço de programação.
- **Usuários avançados** – Interagem com o sistema sem escrever programas. Em vez disso, eles formam suas requisições em uma linguagem de consulta de banco de dados. Eles submetem cada uma dessas consultas a um **processador de consulta**, cuja função é desmembrar as instruções DML em instruções que o gerenciador de armazenamento entenda. Estão nesta categoria os analistas que submetem consultas para explorar dados no banco de dados.
- **Usuários especializados** – São usuários avançados que escrevem aplicações de banco de dados especializadas que não se encaixam na estrutura de processamento de dados tradicional. Entre essas aplicações estão os sistemas de projeto auxiliados por computador, sistemas de base de conhecimentos e sistemas especialistas, sistemas que armazenam dados com tipos de dados complexos (por exemplo, dados gráficos e dados de áudio) e sistemas de modelagem de ambiente. O Capítulo 9 aborda várias dessas aplicações.

Administrador de banco de dados

Uma das principais razões para usar DBMSs é ter um controle central sobre os dados e os programas que os acessam. Uma pessoa que tem esse controle central sobre o sistema é chamada de **administrador de banco de dados (DBA)**. As funções de um DBA incluem:

- **Definição de esquema**. O DBA cria o esquema de banco de dados original executando um conjunto de instruções de definição de dados na DDL.
- **Estrutura de armazenamento e definição de método de acesso**.
- **Modificação de esquema e de organização física**. O DBA realiza mudanças no esquema e na organização física para refletir as alterações das necessidades da empresa, ou para modificar a organização física de modo a melhorar o desempenho.
- **Concessão de autorização para acesso a dados**. Concedendo diferentes tipos de autorização, o administrador de banco de dados pode controlar que partes do banco de dados os vários usuários podem acessar. As informações de autorização são mantidas em uma estrutura de sistema especial que o sistema de banco de dados consulta toda vez que alguém tenta acessar os dados no sistema.
- **Manutenção de rotina**. Exemplos da manutenção de rotina do administrador de banco de dados são:
  - Realizar backups periódicos do banco de dados, sejam em fitas ou em servidores remotos, para prevenir perda de dados no caso de acidentes, como incêndio, inundação etc.
  - Garantir que haja suficiente espaço livre em disco para operações normais e aumentar o espaço em disco conforme necessário.

- Monitorar tarefas sendo executadas no banco de dados e assegurar que o desempenho não seja comprometido por tarefas muito onerosas submetidas por alguns usuários.

História dos sistemas de banco de dados

O processamento de dados tem impulsionado o crescimento dos computadores desde os primeiros dias dos computadores comerciais. Na verdade, a automação das tarefas de processamento de dados já existia antes mesmo dos computadores. Cartões perfurados, inventados por Herman Hollerith, foram usados no início do século XX para registrar dados de censo nos Estados Unidos, e sistemas mecânicos foram usados para processar os cartões e tabular os resultados. Mais tarde, os cartões perfurados passaram a ser amplamente usados como um meio de inserir dados em computadores.

As técnicas para armazenamento e processamento de dados evoluíram ao longo dos anos:

- **Década de 1950 e início da década de 1960:** as fitas magnéticas foram desenvolvidas para armazenamento de dados. As tarefas de processamento de dados, como folha de pagamento, eram automatizadas, com dados armazenados em fitas. O processamento consistia em ler dados de uma ou mais fitas e escrevê-los em uma nova fita. Os dados também podiam ser inseridos por decks de cartão perfurado e enviados para saída em impressoras. Por exemplo, os aumentos salariais eram processados inserindo os aumentos em cartões perfurados e lendo o deck de cartão perfurado em sincronia com uma fita contendo os detalhes de salário mestres. Os registros precisavam estar na mesma ordem de classificação. Os aumentos salariais seriam acrescentados ao salário lido da fita mestre e escritos em uma nova fita, que se tornaria a nova fita mestre.

  As fitas (e os decks de cartão) podiam ser lidas apenas seqüencialmente, e os tamanhos de dados eram muito maiores do que a memória principal; portanto, os programas de processamento de dados eram obrigados a processar dados em uma ordem específica, lendo e mesclando dados de fitas e decks de cartão.
- **Final da década de 1960 e década de 1970:** o uso difundido dos discos rígidos no final da década de 1960 mudou drasticamente o cenário do processamento de dados, já que os discos rígidos permitiam acesso direto aos dados. A posição dos dados no disco era indiferente, uma vez que qualquer local no disco podia ser acessado em apenas dezenas de milissegundos. Os dados, portanto, estavam livres da tirania da seqüencialidade. Com os discos, podiam ser criados bancos de dados em rede e hierárquicos que permitissem que estruturas de dados, como listas e árvores, fossem armazenadas no disco. Os programadores podiam construir e manipular essas estruturas de dados.

  Um documento revolucionário de Codd [1970] definiu o modelo relacional e os métodos procedurais de consultar dados no modelo relacional, dando origem aos bancos de dados relacionais. A simplicidade do modelo relacional e a possibilidade de ocultar completamente os detalhes de implementação do programador eram propostas realmente tentadoras. Mais tarde, Codd ganhou o prestigiado prêmio da Association of Computing Machinery Turing pelo seu trabalho.
- **Década de 1980:** embora academicamente interessante, inicialmente o modelo relacional não era usado na prática, devido às suas desvantagens de desempenho percebidas; os bancos de dados relacionais não podiam se igualar aos bancos de dados de rede e hierárquicos existentes. Isso mudou com o System R, um projeto inovador da IBM Research que desenvolvia técnicas para a construção de um sistema de banco de dados relacional eficiente. Excelentes descrições do System R são fornecidas por Astrahan *et al.* [1976] e Charberlin *et al.* [1981]. O protótipo totalmente funcional do System R levou ao primeiro produto de banco de dados relacional da IBM, o SQL/DS. Os primeiros sistemas de banco de dados relacionais comerciais, como o IBM DB2, Oracle, Ingres e DEC Rdb, desempenharam um papel fundamental no desenvolvimento de técnicas para processamento eficiente de consultas declarativas. No início da década de 1980, os bancos de dados relacionais haviam se tornado competitivos com os sistemas de banco de dados hierárquicos e de rede, mesmo na área do desempenho. Os bancos de dados relacionais eram tão fáceis de usar que, posteriormente, substituíram os bancos de dados hierárquicos e de rede; os programadores usando esses bancos de dados eram obrigados a lidar com muitos detalhes de implementação de baixo nível e precisavam codificar suas consultas de uma maneira procedural. Mais importante, eles precisavam ter a eficiência em mente quando projetavam seus programas, o que envolvia muito trabalho. Ao contrário, em um banco de dados relacional, quase todas essas tarefas de baixo nível são realizadas automaticamente pelo banco de dados, deixando o programador livre para trabalhar em um nível lógico. Desde que alcançou domínio na década de 1980, o modelo relacional tem reinado supremo entre os modelos de dados.

  A década de 1980 também presenciou muita pesquisa sobre bancos de dados paralelos e distribuídos, bem como um trabalho inicial sobre bancos de dados orientados a objeto.

- **Início da década de 1990:** a linguagem SQL foi projetada principalmente para aplicações de suporte a decisão, que são concentradas na consulta, enquanto a base dos bancos de dados na década de 1980 eram as aplicações de processamento de transação, que são concentradas na atualização. O suporte a decisão e a criação de consulta emergiram novamente como uma importante área de aplicação para bancos de dados. As ferramentas para analisar grandes quantidades de dados experimentaram um elevado crescimento no uso.

  Muitos fornecedores de banco de dados introduziram produtos de banco de dados paralelos nesse período. Os fornecedores de banco de dados também começaram a acrescentar suporte relacional de objeto a seus bancos de dados.
- **Final da década de 1990:** o evento mais importante foi o crescimento explosivo da World Wide Web. Os bancos de dados eram utilizados muito mais extensivamente do que jamais foram. Os sistemas de banco de dados agora tinham de aceitar taxas de processamento de transação muito altas, bem como uma confiabilidade bastante alta e disponibilidade de 24 × 7 (disponibilidade 24 horas por dia, 7 dias por semana, significando nenhum período de inatividade para ações de manutenção programadas). Os sistemas de banco de dados também precisavam aceitar interfaces da Web para dados.
- **Início da década de 2000:** no início da década de 2000, vimos o surgimento da XML e da linguagem de consulta associada, XQuery, como uma nova tecnologia de banco de dados. O júri ainda não tem um veredicto quanto ao papel que a XML desempenhará nos bancos de dados futuros. Nesse período de tempo, também observamos o crescimento das técnicas de "computação autonômica/auto-administração" para reduzir os esforços de administração de sistema.

Resumo

- Um **sistema de gerenciamento de banco de dados (DBMS)** consiste em uma coleção de dados inter-relacionados e um conjunto de programas para acessá-los. Os dados descrevem uma atividade específica.
- O principal objetivo de um DBMS é fornecer um ambiente que seja tanto conveniente quanto eficiente para as pessoas usarem na recuperação e armazenamento de informações.
- Os sistemas de banco de dados estão em toda parte atualmente, e a maioria das pessoas interage, direta ou indiretamente, com bancos de dados muitas vezes todos os dias.
- Os sistemas de banco de dados são projetados para armazenar grandes blocos de informação. O gerenciamento de dados envolve tanto a definição das estruturas para o armazenamento de informações como a provisão dos mecanismos para a manipulação das informações. Além disso, o sistema de banco de dados precisa fornecer a segurança das informações armazenadas, no caso de falhas do sistema ou tentativas de acesso não-autorizado. Se os dados forem compartilhados entre vários usuários, o sistema precisa evitar possíveis resultados anômalos.
- Uma importante finalidade de um sistema de banco de dados é fornecer aos usuários uma visão abstrata dos dados. Isto é, o sistema oculta certos detalhes de como os dados são armazenados e mantidos.
- Apoiando a estrutura de um banco de dados está o **modelo de dados**: uma coleção de ferramentas conceituais para descrever dados, suas relações, sua semântica e suas restrições.
- Uma **linguagem de manipulação de dados (DML)** permite aos usuários acessar ou manipular dados. As DMLs não procedurais, que exigem que o usuário especifique apenas quais dados são necessários, sem especificar exatamente como obtê-los, são amplamente usadas hoje.
- Uma **linguagem de definição de dados (DDL)** é uma linguagem para especificar o esquema de banco de dados, bem como outras propriedades dos dados.
- O modelo de dados relacional é o modelo mais utilizado para armazenar dados em bancos de dados. Outros modelos de dados são o modelo orientado a objeto, o modelo relacional de objetos e os modelos de dados semi-estruturados.
- O projeto de banco de dados envolve basicamente o projeto do esquema de banco de dados. O modelo de dados de entidade/relacionamento (E-R) é bastante utilizado para projeto de banco de dados. Ele fornece uma representação gráfica conveniente para ver dados, relações e restrições.
- Um sistema de banco de dados possui vários subsistemas.
  - O subsistema **gerenciador de armazenamento** fornece a interface entre os dados de baixo nível armazenados no banco de dados e os programas de aplicação e consultas submetidas ao sistema.
  - O subsistema **processador de consulta** compila e executa instruções DDL e DML.
- O **gerenciamento de transação** garante que o banco de dados permanece em um estado consistente (correto) apesar de falhas do sistema. O gerenciador de transação assegura que as execuções de transação concorrentes sejam realizadas sem conflitos.
- As aplicações de banco de dados normalmente são divididas em uma parte de front-end que é executada em máquinas clientes e uma parte que é executada no back-end. Nas arquiteturas de duas camadas, o front-end se comunica diretamente com um banco de dados

sendo executado no back-end. Nas arquiteturas de três camadas, a parte do back-end é, ela mesma, dividida em um servidor de aplicação e um servidor de banco de dados.
- Os usuários de banco de dados podem ser categorizados em várias classes, e cada uma geralmente utiliza um tipo diferente de interface com o banco de dados.

Termos de revisão

- Sistema de gerenciamento de banco de dados (DBMS)
- Aplicações de sistemas de banco de dados
- Sistemas de arquivo
- Inconsistência de dados
- Restrições de consistência
- Visões de dados
- Abstração de dados
- Instância de banco de dados
- Esquema
  - Esquema de banco de dados
  - Esquema físico
  - Esquema lógico
- Independência de dados física
- Modelos de dados
  - Modelo de entidade/relacionamento
  - Modelo de dados relacional
  - Modelo de dados orientado a objeto
  - Modelo de dados relacional de objeto
- Linguagens de banco de dados
  - Linguagem de definição de dados
  - Linguagem de manipulação de dados
  - Linguagem de consulta
- Dicionário de dados
- Metadados
- Transações
- Concorrência
- Programa de aplicação
- Administrador de banco de dados (DBA)
- Máquinas cliente e servidora

Exercícios práticos

1.1 Este capítulo descreveu várias vantagens importantes de um sistema de banco de dados. Cite duas desvantagens.

1.2 Liste sete linguagens de programação que sejam procedurais e duas que sejam não procedurais. Que grupo é mais fácil de aprender e usar? Explique sua resposta.

1.3 Relacione seis etapas importantes que você executaria na configuração de um banco de dados para uma determinada organização.

1.4 Considere uma matriz de inteiros bidimensional de tamanho $n \times m$ que deve ser usada em sua linguagem de programação favorita. Usando a matriz como um exemplo, ilustre a diferença (a) entre os três níveis de abstração de dados e (b) entre um esquema e instâncias.

Exercícios

1.5 Liste quatro aplicações que você usou, que mais provavelmente tenha empregado um sistema de banco de dados para armazenar dados persistentes.

1.6 Liste quatro diferenças significantes entre um sistema de processamento de arquivos e um DBMS.

1.7 Explique a diferença entre independência de dados física e lógica.

1.8 Cite cinco responsabilidades de um sistema de gerenciamento de banco de dados. Para cada responsabilidade, explique os problemas que surgiriam se a responsabilidade não fosse delegada.

1.9 Liste pelo menos duas razões por que os sistemas de banco de dados aceitam manipulação de dados usando uma linguagem de consulta declarativa como SQL, em vez de apenas fornecerem uma biblioteca de funções C ou C++ para realizar manipulação de dados.

1.10 Explique que problemas são causados pelo projeto da tabela na Figura 1.5.

1.11 Cite as cinco funções principais de um administrador de banco de dados.

Notas bibliográficas

A seguir, relacionamos livros de finalidade geral, coleções de relatórios e sites sobre bancos de dados. Os capítulos subsequentes fornecem referências para materiais sobre cada tópico descrito neste capítulo.

Codd [1970] é o estudo revolucionário que apresentou o modelo relacional.

Livros abordando os sistemas de banco de dados incluem Abiteboul *et al.* [1995], Date [2003], Elmasri e Navathe [2003], O'Neil [2000], Ramakrishnan e Gehrke [2002], Garcia-Molina *et al.* [2001] e Ullman [1988]. Bernstein e Newcomer [1997] e Gray e Reuter [1993] oferecem livros-texto sobre processamento de transação.

Vários livros contêm coleções de relatórios sobre gerenciamento de banco de dados. Entre esses estão Bancilhon e Buneman [1990], Date [1986], Date [1990], Kim [1995], Zaniolo *et al.* [1997] e Hellerstein e Stonebraker [2005].

Uma análise das realizações no gerenciamento de banco de dados e uma avaliação dos futuros desafios de pesquisa aparecem em Silberschatz *et al.* [1990], Silberschatz *et al.*

[1996], Bernstein *et al.* [1998] e Abiteboul *et al.* [2003]. A home page do ACM Special Interest Group on Management of Data (`www.acm.org/sigmod`) fornece uma variedade de informações sobre pesquisa de banco de dados. Sites de fornecedores de banco de dados (veja a seção "Ferramentas" a seguir) fornecem detalhes sobre seus respectivos produtos.

Ferramentas

Existem muitos sistemas de banco de dados comerciais em uso atualmente. Os principais incluem: IBM DB2 (`www.ibm.com/software/data`), Oracle (`www.oracle.com`), Microsoft SQL Server (`www.microsoft.com/sql`), Informix (`www.informix.com`) (agora pertencente à IBM) e Sybase (`www.sybase.com`). Alguns desses sistemas estão disponíveis gratuitamente para uso pessoal ou não comercial, ou para desenvolvimento, mas não são gratuitos para utilização real.

Também existem diversos sistemas de banco de dados de domínio público, gratuitos; os amplamente usados incluem MySQL (`www.mysql.com`) e PostgreSQL (`www.postgresql.org`).

PARTE 1

Bancos de dados relacionais

Um modelo de dados é uma coleção de ferramentas conceituais para descrever dados, relações de dados, semântica de dados e restrições de consistência. Nesta parte focalizamos o modelo relacional.

O modelo relacional usa um conjunto de tabelas para representar tanto os dados quanto as relações entre eles. Sua simplicidade conceitual levou ao seu amplo uso; hoje, a grande maioria dos produtos de banco de dados é baseada no modelo relacional. Na Parte 9, descrevemos quatro sistemas de banco de dados relacionais amplamente usados.

Embora o modelo relacional, abordado no Capítulo 2, descreva dados nos níveis lógico e de visão, ele é um modelo de dados de nível inferior, se comparado com o modelo entidade-relacionamento discutido na Parte 2.

Para tornarmos os dados de um banco de dados relacional disponíveis aos usuários, precisamos tratar de vários problemas. Um deles é como os usuários especificam requisições de dados: qual das várias linguagens de consulta eles usam? Os Capítulos 3 e 4 abordam a SQL, que é a linguagem de consulta mais utilizada atualmente. O Capítulo 5 primeiramente discute duas linguagens de consulta formais, o cálculo relacional de tupla e o cálculo relacional de domínio, que são linguagens de consulta declarativas baseadas em lógica matemática. Essas duas linguagens formais são a base de mais duas linguagens amigáveis, QBE e Datalog, que estudamos mais adiante no Capítulo 5.

Outro problema é a integridade e a proteção de dados; os bancos de dados precisam proteger os dados de danos causados por ações do usuário, sejam intencionais ou não. O componente de manutenção de integridade de um banco de dados garante que as atualizações não violem as restrições de integridade que foram especificadas nos dados. O componente de proteção de um banco de dados inclui controle de acesso para restringir as ações permissíveis para cada usuário. O Capítulo 4 aborda os problemas da integridade e da proteção. Os problemas de integridade e proteção estão presentes independentemente do modelo de dados, mas, para exatidão, os estudaremos no contexto do modelo relacional.

CAPÍTULO 2

Modelo relacional

O modelo relacional é hoje o principal modelo de dados para aplicações comerciais de processamento de dados. Ele conquistou sua posição de destaque devido à sua simplicidade, que facilita o trabalho do programador, comparado com os modelos de dados anteriores, como o modelo de rede ou o modelo hierárquico.

Neste capítulo, primeiro estudamos os fundamentos do modelo relacional. Em seguida, descrevemos a álgebra relacional, usada para especificar requisições de informações. A álgebra relacional não é amigável ao usuário, mas serve como a base formal para linguagens de consulta amigáveis que estudaremos mais tarde, incluindo a amplamente usada linguagem de consulta SQL, que abordamos em detalhes nos Capítulos 3 e 4.

Existe uma importante teoria para os bancos de dados relacionais. Neste capítulo, estudamos a parte dessa teoria que lida com consultas. Nos Capítulos 6 e 7, examinaremos aspectos da teoria de banco de dados relacional que ajudam no projeto dos esquemas de banco de dados relacionais, enquanto nos Capítulos 13 e 14, discutimos aspectos da teoria que lida com o processamento eficiente de consultas.

Estrutura dos bancos de dados relacionais

Um banco de dados relacional consiste em uma coleção de tabelas, cada uma com um nome único atribuído. Uma linha em uma tabela representa uma *relação* entre um conjunto de valores. Informalmente, uma tabela é um conjunto de entidades, e uma linha é uma entidade, como discutimos no Capítulo 1. Como uma tabela é uma coleção dessas relações, existe uma íntima correspondência entre o conceito de *tabela* e o conceito matemático de *relação*, do qual o modelo de dados relacional extrai seu nome. A seguir, apresentamos o conceito de relação.

Neste capítulo, vamos usar várias relações diferentes para ilustrar os diversos conceitos que sustentam o modelo de dados relacional. Essas relações representam parte de uma instituição bancária. Para simplificar nossa apresentação, elas podem não corresponder à maneira real como um banco de dados bancário pode estar estruturado. Discutiremos em detalhes os critérios para a adequabilidade das estruturas relacionais nos Capítulos 6 e 7.

Estrutura básica

Considere a tabela *conta* da Figura 2.1. Ela possui três cabeçalhos de coluna: *número_conta*, *nome_agência* e *saldo*. Segundo a terminologia do modelo relacional, chamamos esses cabeçalhos de **atributos**. Para cada atributo, existe um conjunto de valores permitidos, que é o **domínio** desse atributo. Para o atributo *nome_agência*, por exemplo, o domínio é o conjunto de todos os nomes de agência. Seja D_1 o conjunto de todos os números de conta, D_2 o conjunto de todos os nomes de agência e D_3 o conjunto de todos os saldos. Qualquer linha de *conta* precisa consistir em uma tupla de 3 (v_1, v_2 e v_3), onde v_1 é um número de conta (ou seja, v_1 está no domínio D_1), v_2 é um nome de agência (ou seja, v_2 está no domínio D_2) e v_3 é um saldo (ou seja, v_3 está no domínio D_3). Em geral, *conta* conterá apenas um subconjunto do conjunto de todas as linhas possíveis. Portanto, *conta* é um subconjunto de

$$D_1 \times D_2 \times D_3$$

Em geral, uma **tabela** de n atributos precisa ser um subconjunto de

$$D_1 \times D_2 \times \ldots \times D_{n-1} \times D_n$$

número_conta	*nome_agência*	*saldo*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

Figura 2.1 A relação *conta*.

Os matemáticos definem uma **relação** como sendo um subconjunto de um produto cartesiano de uma lista de domínios. Essa definição corresponde quase exatamente à nossa definição de *tabela*. A única diferença é que atribuímos nomes a atributos, enquanto os matemáticos se baseiam em "nomes" numéricos, usando o inteiro 1 para denotar o atributo cujo domínio aparece em primeiro na lista de domínios, 2 para o atributo cujo domínio aparece em segundo e assim por diante. Como as tabelas são basicamente relações, usaremos os termos matemáticos **relação** e **tupla** no lugar dos termos **tabela** e **linha**. Uma **variável de tupla** é uma variável que significa uma tupla; em outras palavras, uma variável de tupla é uma variável cujo domínio é o conjunto de todas as tuplas.

Na relação *conta* da Figura 2.1, existem sete tuplas. Faça a variável de tupla t se referir à primeira tupla da relação. Usamos a notação $t[número\_conta]$ para denotar o valor de t no atributo *número_conta*. Portanto, $t[número\_conta] = \text{"A-101"}$ e $t[nome\_agência] = \text{"Downtown"}$. Alternativamente, podemos escrever $t[1]$ para denotar o valor da tupla t no primeiro atributo (*número_conta*), $t[2]$ para denotar *nome_agência* e assim por diante. Como uma relação é um conjunto de tuplas, usamos a notação matemática de $t \ r$ para denotar que a tupla t está na relação r.

A ordem em que as tuplas aparecem em uma relação é irrelevante, já que uma relação é um *conjunto* de tuplas. Portanto, quer as tuplas de uma relação sejam listadas em or
dem, como na Figura 2.1, ou estejam desordenadas, com
na Figura 2.2, as relações nas duas figuras são as mesmas
pois ambas contêm o mesmo conjunto de tuplas.

Exigimos que, para todas as relações, os domínios de to
dos os atributos de r sejam atômicos. Um domínio é atômi
co se os elementos do domínio são considerados unidade
indivisíveis. Por exemplo, o conjunto dos inteiros é um do
mínio atômico, mas o conjunto de todos os conjuntos d
inteiros é um domínio não atômico. A diferença é que nor
malmente não consideramos os inteiros como tendo sub
conjuntos, mas consideramos conjuntos de inteiros com
tendo subconjuntos – a saber, os inteiros que compõem
conjunto. O aspecto importante não é qual é o domínio en
si, mas como usamos elementos de domínio em nosso ban
co de dados. Os domínios de *saldo* e *nome_agência*, por ou
tro lado, certamente devem ser diferentes. Talvez seja me
nos claro se *nome_cliente* e *nome_agência* devam ter o mes
mo domínio. No nível físico, os nomes de cliente e os no
mes de argumento são strings de caractere. Entretanto, n
nível lógico, podemos querer que *nome_cliente* e *nome_agên
cia* tenham domínios distintos.

Um valor de domínio que é um membro de qualquer do
mínio possível é o valor **nulo**, o que significa que o valor
desconhecido ou não existe. Por exemplo, suponha que in
cluamos o atributo *número_telefone* na relação *cliente*. Pod

número_conta	*nome_agência*	*saldo*
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

Figura 2.2 A relação *conta* com tuplas desordenadas.

nome_agência	*cidade_agência*	*ativo*
Brighton	Brooklyn	7.100.000
Downtown	Brooklyn	9.000.000
Mianus	Horseneck	400.000
North Town	Rye	3.700.000
Perryridge	Horseneck	1.700.000
Pownal	Bennington	3.000.000
Redwood	Palo Alto	2.100.000
Round Hill	Horseneck	8.000.000

Figura 2.3 A relação *agência*.

ser que um cliente não tenha um número de telefone, ou que o número de telefone não esteja disponível. Então, precisaríamos recorrer aos valores nulos para indicar que o valor é desconhecido ou não existe. Mais adiante, veremos que os valores nulos causam diversas dificuldades quando acessamos ou atualizamos o banco de dados, e, portanto, devem ser evitados se possível. Inicialmente, consideraremos que os valores nulos estão ausentes e na seção "Valores nulos" descreveremos o efeito de nulos em diferentes operações.

Esquema de banco de dados

Quando falamos de bancos de dados, precisamos diferenciar entre o **esquema de banco de dados**, que é o projeto lógico do banco de dados, e a **instância de banco de dados**, que é o instantâneo dos dados no banco de dados em um determinado instante no tempo.

O conceito de uma relação corresponde à noção de linguagem de programação de uma variável. O conceito de um **esquema de relação** corresponde à noção de linguagem de programação da definição de tipo.

É conveniente atribuir um nome a um esquema de relação, exatamente como atribuímos nomes a definições de tipo nas linguagens de programação. Adotamos a convenção de usar nomes em letras minúsculas para relações e nomes iniciando com uma letra maiúscula para esquemas de relação. Segundo essa notação, usamos *Esquema_conta* para denotar o esquema de relação para a relação *conta*. Assim,

Esquema_conta = (*número_conta, nome_agência, saldo*)

Denotamos o fato de que *conta* é uma relação em *Esquema_conta* por

conta(*Esquema_conta*)

Em geral, um esquema de relação consiste em uma lista dos atributos e seus domínios correspondentes. Não nos preocuparemos com a definição exata do domínio de cada atributo até discutirmos a linguagem SQL nos Capítulos 3 e 4.

O conceito de uma **instância de relação** corresponde à noção da linguagem de programação de um valor de uma variável. O valor de uma determinada variável pode mudar com o tempo; da mesma forma, o conteúdo de uma instância de relação pode mudar com o tempo conforme a relação é atualizada. Entretanto, em geral, dizemos simplesmente "relação" quando, na verdade, queremos nos referir a "instância de relação".

Como um exemplo de uma instância de relação, considere a relação *agência* da Figura 2.3. O esquema para essa relação é

Esquema_agência = (*nome_agência, cidade_agência, ativo*)

Observe que o atributo *nome_agência* aparece em *Esquema_agência* e em *Esquema_conta*. Essa duplicação não é uma coincidência. Usar atributos comuns em esquemas de relação é uma maneira de relacionar tuplas de relações diferentes. Por exemplo, suponha que desejamos encontrar as informações sobre todas as contas mantidas nas agências localizadas em Brooklyn. Então, para cada uma dessas agências, olhamos na relação *conta* a fim de encontrar as informações sobre as contas mantidas nessa agência.

Vamos continuar com nosso exemplo de banco. Precisamos de uma relação para descrever informações sobre clientes. O esquema de relação é

Esquema_cliente = (*nome_cliente, rua_cliente, cidade_cliente*)

A Figura 2.4 mostra uma relação *cliente* de exemplo (*Esquema_cliente*). Note que omitimos o atributo *id_cliente* utilizado no Capítulo 1 porque, agora, queremos ter esquemas de relação menores em nosso exemplo funcional de um banco de dados de banco. Consideramos que o nome de

nome_cliente	*rua_cliente*	*cidade_cliente*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Harrison
Johnson	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Figura 2.4 A relação *cliente*.

cliente identifica de maneira única um cliente – é claro, isso pode não ser verdade no mundo real, mas a suposição torna nossos exemplos muito mais fáceis de ler. Em um banco de dados real, o *id_cliente* (que poderia ser um número de *seguro_social* ou um identificador gerado pelo banco) serviria para identificar os clientes unicamente.

Também precisamos de uma relação para descrever a associação entre clientes e contas. O esquema de relação para descrever essa associação é

Esquema_depositante = (*nome_cliente*, *número_conta*)

A Figura 2.5 mostra uma relação *depositante* de exemplo (*Esquema_depositante*).

Pareceria que, para nosso exemplo de banco, poderíamos ter apenas um esquema, em vez de vários. Ou seja, é mais fácil para um usuário pensar em termos de um esquema de relação, em vez de vários. Suponha que usamos apenas uma relação para nosso exemplo, com o esquema

(*nome_agência*, *cidade_agência*, *ativo*, *nome_cliente*, *rua_cliente*, *cidade_cliente*, *número_conta*, *saldo*)

Observe que, se um cliente possui várias contas, precisamos listar seu endereço uma vez para cada conta. Ou seja, precisamos repetir certas informações várias vezes. Essa repetição é um desperdício e é evitada com o uso de várias relações, como em nosso exemplo.

Além disso, se uma agência não tiver qualquer conta (digamos, uma agência recém-aberta, que ainda não possui clientes), não podemos construir uma tupla completa na relação simples anterior, pois ainda não existem dados concernentes a *cliente* e *conta* disponíveis. Para representar tuplas incompletas, precisamos usar valores *nulos*, que significam que o valor é desconhecido ou inexistente. Portanto, em nosso exemplo, os valores para *nome_cliente*, *rua_cliente* etc. precisam ser nulos. Usando várias relações, podemos representar as informações de agência para um banco sem clientes sem usar valores nulos. Simplesmente usamos uma tupla em *Esquema_agência* para repre-

nome_cliente	*número_conta*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Figura 2.5 A relação *depositante*.

número_empréstimo	*nome_agência*	*quantia*
L-11	Round Hill	900
L-11	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figura 2.6 A relação *empréstimo*.

sentar as informações sobre a agência e criamos tuplas nos outros esquemas apenas quando as informações apropriadas se tornarem disponíveis.

No Capítulo 7, estudaremos critérios para nos ajudar a decidir quando um conjunto de esquemas de relação é mais apropriado do que outro, em termos de repetição de informação e da existência de valores nulos. Por enquanto, consideraremos que os esquemas de relação são fornecidos.

Incluímos duas relações adicionais para descrever dados sobre empréstimos mantidos nas várias agências do banco:

Esquema_empréstimo = (*número_empréstimo, nome_agência, conta*)
Esquema_tomador = (*nome_cliente, número_empréstimo*)

As Figuras 2.6 e 2.7, respectivamente, mostram as relações de exemplo *empréstimo* (*Esquema_empréstimo*) e *tomador* (*Esquema_tomador*).

Os esquemas de relação correspondem ao conjunto de tabelas que podemos gerar pelo método esboçado na seção "Projeto de um banco de dados do primeiro capítulo". Observe que a relação *cliente* pode conter informações sobre clientes que não possuem nem uma conta nem um empréstimo no banco. O projeto de banco descrito aqui servirá como nosso principal exemplo neste capítulo. Ocasionalmente, precisaremos introduzir esquemas de relação adicionais para ilustrar aspectos específicos.

Chaves

É preciso ter uma maneira de especificar como as tuplas dentro de uma determinada relação são distinguidas. Isso é expresso em termos de seus atributos. Ou seja, os valores de atributo de uma tupla precisam ser tais que possam *identificar unicamente* a tupla. Em outras palavras, nenhum par de tuplas em uma relação pode ter exatamente o mesmo valor para todos os atributos.

Uma **superchave** é um conjunto de um ou mais atributos que, tomados coletivamente, nos permite identificar unicamente uma tupla na relação. Por exemplo, o atributo *id_cliente* da relação *cliente* é suficiente para distinguir uma tupla *cliente* de outra. Portanto, *id_cliente* é uma superchave. Da mesma forma, a combinação de *nome_cliente* e *id_cliente* é uma superchave para a relação *cliente*. O atributo *nome_cliente* de *cliente* não é uma superchave porque várias pessoas poderiam ter o mesmo nome.

O conceito de uma superchave não é suficiente para nossos propósitos, já que, como dissemos, uma superchave

nome_cliente	*número_empréstimo*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figura 2.7 A relação *tomador*.

pode conter atributos estranhos. Se K é uma superchave, então também será uma superchave qualquer superconjunto de K. Normalmente, o que nos interessa são superchaves para as quais nenhum subconjunto apropriado é uma superchave. Essas superchaves mínimas são chamadas **chaves candidatas**.

É possível que vários conjuntos diferentes de atributos sirvam como uma chave candidata. Suponha que uma combinação de *nome_cliente* e *rua_cliente* seja suficiente para distinguir entre membros da relação *cliente*. Então, {*id_cliente*} e {*nome_cliente*, *rua_cliente*} são chaves candidatas. Embora os atributos *id_cliente* e *nome_cliente* juntos possam distinguir tuplas *cliente*, sua combinação não forma uma chave candidata, já que o atributo *id_cliente* sozinho é uma chave candidata.

Usaremos o termo **chave primária** para denotar uma chave candidata que é escolhida pelo projetista de banco de dados como o principal meio de identificar tuplas dentro de uma relação. Uma chave (seja primária, candidata ou superchave) é uma propriedade da relação inteira, e não das tuplas individuais. Nenhum par de tuplas na relação pode ter o mesmo valor nos atributos de chave ao mesmo tempo. A designação de uma chave representa uma restrição constante na empresa real sendo modelada.

As chaves candidatas precisam ser escolhidas com cuidado. Como salientamos, o nome de uma pessoa obviamente não é suficiente, pois pode haver muitas pessoas com o mesmo nome. Nos Estados Unidos, o atributo número de seguro social de uma pessoa seria uma chave candidata. Uma vez que os não residentes nos Estados Unidos normalmente não possuem números de seguro social, as empresas internacionais precisam gerar seus próprios identificadores únicos. Uma alternativa é usar alguma combinação única de outros atributos como uma chave.

A chave primária deve ser escolhida de modo que seus valores de atributo nunca, ou muito raramente, sejam modificados. Por exemplo, o campo Endereço de uma pessoa não deve ser parte da chave primária, já que pode mudar. Os números de seguro social, por outro lado, oferecem a garantia de nunca mudar. Os identificadores únicos gerados pelas empresas normalmente não mudam, exceto se duas empresas se fundirem; nesse caso, o mesmo identificador pode ter sido emitido pelas duas empresas, e uma relocação dos identificadores pode ser necessária para garantir que eles sejam únicos.

Formalmente, façamos R ser um esquema de relação. Se dissermos que um subconjunto K de R é uma *superchave* para R, estamos restringindo a consideração às relações $r(R)$ em que nenhum par de tuplas distintas tem os mesmos valores de todos os atributos em K. Ou seja, se t_1 e t_2 estão em r e $t_1 \neq t_2$, então $t_1[K] \neq t_2[K]$.

Um esquema de relação, digamos, r_1, pode incluir entre seus atributos a chave primária de outro esquema de relação, digamos, r_2. Esse atributo é chamado de **chave estrangeira** de r_1, referenciando r_2. A relação r_1 também é chamada de **relação referenciadora** da dependência da chave estrangeira, e r_2 é chamada de **relação referenciada** da chave estrangeira. Por exemplo, o atributo *nome_agência* em *Esquema_conta* é uma chave estrangeira de *Esquema_conta* referenciando *Esquema_agência*, já que *nome_agência* é a chave primária de *Esquema_agência*. Em qualquer instância de banco de dados, dada qualquer tupla, digamos, t_a, da relação *conta*, precisa haver alguma tupla, digamos, t_b, na relação *agência*, tal que o valor do atributo *nome_agência* de t_a seja o mesmo que o valor da chave primária, *nome_agência*, de t_b.

É comum listar os atributos de chave primária de um esquema de relação antes dos outros atributos; por exemplo, o atributo *nome_agência* de *Esquema_agência* é listado primeiro, já que ele é a chave primária.

O esquema de banco de dados, juntamente com a chave primária e as dependências da chave estrangeira, pode ser representado graficamente pelos **diagramas de esquema**. A Figura 2.8 mostra o diagrama de esquema para nossa instituição bancária. Cada relação aparece como uma caixa,

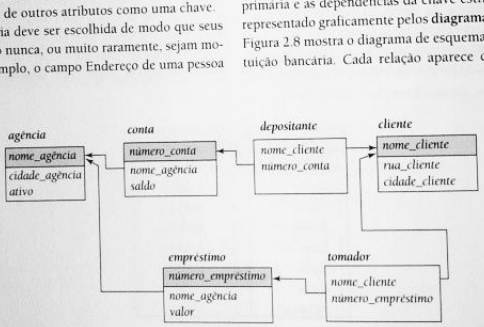


Figura 2.8 Diagrama de esquema para a instituição bancária.

com os atributos listados dentro e o nome da relação em cima. Se houver atributos de chave primária, uma linha horizontal atravessa a caixa, com os atributos de chave primária listados acima da linha em cinza. As dependências de chave estrangeira aparecem como setas indo dos atributos de chave estrangeira da relação referenciadora até a chave primária da relação referenciada.

Muitos sistemas de banco de dados fornecem ferramentas de projeto com uma interface gráfica com o usuário (GUI) para criar diagramas de esquema.

Linguagens de consulta

Uma **linguagem de consulta** é uma linguagem em que um usuário requisita informações do banco de dados. Essas linguagens normalmente estão em um nível mais alto que o de uma linguagem de programação padrão. As linguagens de consulta podem ser categorizadas como procedurais ou não procedurais. Em uma **linguagem procedural**, o usuário instrui o sistema a realizar uma sequência de operações no banco de dados para calcular o resultado desejado. Em uma **linguagem não procedural**, o usuário descreve as informações desejadas sem fornecer um procedimento específico para obtê-las.

A maioria dos sistemas de banco de dados relacionais comerciais oferece uma linguagem de consulta que inclui elementos dos métodos procedural e não procedural. Estudaremos a amplamente usada linguagem de consulta SQL nos Capítulos 3 e 4. O Capítulo 5 trata das linguagens de consulta QBE e Datalog, este último, uma linguagem que se assemelha à linguagem de programação Prolog.

Existem diversas linguagens de consulta "puras": A álgebra relacional é procedural, enquanto o cálculo relacional de tupla e o cálculo relacional de domínio são não procedurais. Essas linguagens de consulta são concisas e formais, e não possuem o "açúcar sintático" das linguagens comerciais, mas elas ilustram as técnicas fundamentais para extrair dados do banco de dados.

Neste capítulo, examinamos em grande detalhe a linguagem de álgebra relacional (no Capítulo 5 abordamos as linguagens de cálculo relacional de tupla e de cálculo relacional de domínio). A álgebra relacional consiste em um conjunto de operações que usam uma ou duas relações com entrada e produzem uma nova relação como resultado.

As operações fundamentais na álgebra relacional são *seleção*, *projeção*, *união*, *diferença de conjuntos*, *produto cartesiano* e *renomeação*. Além das operações fundamentais, há várias outras operações – interseção de conjuntos, junção natural, divisão e atribuição. Definiremos essas operações em termos das operações fundamentais.

Inicialmente, voltaremos nossa atenção somente para as consultas. Entretanto, uma linguagem de manipulação de dados completa inclui não apenas uma linguagem de consulta, mas também uma linguagem para modificação de banco de dados. Essas linguagens incluem comandos para inserir e excluir tuplas, bem como comandos para modificar partes de tuplas existentes. Examinaremos a modificação de banco de dados após completarmos nossa discussão sobre consultas.

Operações fundamentais da álgebra relacional

As operações seleção, projeção e renomeação são chamadas de operações *unárias*, pois operam em uma relação. As outras três operações operam em pares de relações e, portanto, são chamadas de operações *binárias*.

A operação seleção

A operação **seleção** seleciona tuplas que satisfazem um determinado predicado. Usamos a letra grega sigma (σ) para denotar a seleção. O predicado aparece como um subscrito de σ. A relação de argumento está entre parênteses após o σ. Portanto, para selecionar essas tuplas da relação *empréstimo*, onde a agência é "Perryridge", escrevemos

$$\sigma_{nome\_agência = \text{"Perryridge"}}(empréstimo)$$

Se a relação *empréstimo* é como mostra a Figura 2.6, então a relação que resulta da consulta anterior é como mostra a Figura 2.9.

Podemos encontrar todas as tuplas em que a quantia emprestada é maior do que $1200 escrevendo

$$\sigma_{quantia > 1200}(empréstimo)$$

Em geral, permitimos comparações usando $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ no predicado da seleção. Além disso, podemos combinar vários predicados em um predicado maior usando os co-

número_empréstimo	*nome_agência*	*quantia*
L-15	Perryridge	1500
L-16	Perryridge	1300

Figura 2.9 Resultado de $\sigma_{nome\_agência = \text{"Perryridge"}}(empréstimo)$.

nectivos *e* ($\wedge$), *ou* ($\vee$) e *não* ($\neg$). Portanto, para encontrar as tuplas pertencentes a empréstimos de mais de $1.200 feitos pela agência Perryridge, escrevemos

$$\sigma_{nome\_agência = \text{"Perryridge"} \wedge quantia > 1200}(empréstimo)$$

O predicado da seleção pode incluir comparações entre dois atributos. Para ilustrar, considere a relação *gerente_empréstimo*, que consiste em três atributos: *nome_cliente*, *nome_banqueiro* e *número_empréstimo*, que informa que um banqueiro específico é o gerente de um empréstimo pertencente a algum cliente. Para encontrar todos os clientes com o mesmo nome que seu gerente de empréstimo, podemos escrever

$$\sigma_{nome\_cliente = nome\_banqueiro}(gerente\_empréstimo)$$

A operação projeção

Suponha que se deseje listar todos os números de empréstimo e a quantia dos empréstimos, mas o nome da agência não importa. A operação **projeção** permite produzir essa relação. A operação projeção é uma operação unária que retorna sua relação de argumento, com certos atributos omitidos. Como a relação é um conjunto, quaisquer linhas duplicadas são eliminadas.

A projeção é indicada pela letra grega pi (Π). Listamos os atributos que desejamos que apareçam no resultado como um subscrito de Π. A relação de argumento segue entre parênteses. Escrevemos a consulta para listar todos os números de empréstimo e a quantia do empréstimo como

$$\Pi_{número\_empréstimo,\ quantia}(empréstimo)$$

A Figura 2.10 mostra a relação que resulta dessa consulta.

Composição das operações relacionais

É importante atentar para o fato de o resultado de uma operação relacional ser, ele mesmo, uma relação. Considere a consulta mais complexa "Encontre os clientes que moram em Harrison." Escrevemos:

$$\Pi_{nome\_cliente}(\sigma_{cidade\_cliente = \text{"Harrison"}}(cliente))$$

Observe que, em vez de fornecer o nome de uma relação como o argumento da operação de projeção, fornecemos uma expressão que resulta em uma relação.

Em geral, como o resultado de uma operação de álgebra relacional é do mesmo tipo (relação) de suas entradas, as operações de álgebra relacional podem ser compostas juntas em uma **expressão de álgebra relacional**. Compor operações de álgebra relacional em expressões de álgebra relacional é o mesmo que compor operações aritméticas (como $+$, $-$, $*$ e $\div$) em expressões aritméticas. Estudaremos a definição formal das operações de álgebra relacional na seção "Definição formal da álgebra relacional".

A operação união

Considere uma consulta para encontrar os nomes de todos os clientes do banco que possuem uma conta ou um empréstimo, ou ambos. Observe que a relação *cliente* não contém a informação, já que um cliente não precisa ter nem uma conta nem um empréstimo no banco. Para responder a essa consulta, precisamos das informações na relação *depositante* (Figura 2.5) e na relação *tomador* (Figura 2.7). Sabemos como encontrar os nomes de todos os clientes com um empréstimo no banco:

$$\Pi_{nome\_cliente}(tomador)$$

Também sabemos como encontrar os nomes de todos os clientes com uma conta no banco:

$$\Pi_{nome\_cliente}(depositante)$$

Para responder à consulta, precisamos da **união** desses dois conjuntos; ou seja, precisamos que todos os nomes de

número_empréstimo	*quantia*
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Figura 2.10 Número e quantia do empréstimo.

nome_cliente
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

Figura 2.11 Nomes de todos os clientes que têm um empréstimo ou uma conta.

clientes apareçam em alguma ou em ambas as relações. Encontramos esses dados pela operação binária união, indicada, como na teoria dos conjuntos, por ∪. Assim, a expressão aninhada é

$$\Pi_{nome\_cliente}(tomador) \cup \Pi_{nome\_cliente}(depositante)$$

A relação resultante dessa consulta aparece na Figura 2.11. Repare que existem 10 tuplas no resultado, ainda que haja sete tomadores distintos e seis depositantes. Essa aparente discrepância ocorre porque Smith, Jones e Hayes são tomadores e também depositantes. Como as relações são conjuntos, os valores duplicados são eliminados.

Observe que, em nosso exemplo, tomamos a união de dois conjuntos, ambos consistindo nos valores *nome_cliente*. Em geral, precisamos garantir que as uniões sejam tomadas entre relações *compatíveis*. Por exemplo, não faria sentido tomar a união da relação *empréstimo* com a relação *tomador*. A primeira é uma relação de três atributos; a segunda é uma relação de dois. Além disso, considere uma união de um conjunto de nomes de clientes e um conjunto de cidades. Esse tipo de união não faria sentido na maioria das situações. Portanto, para que uma operação de união $r \cup s$ seja válida, exigimos que duas condições sejam satisfeitas:

1. As relações r e s precisam ser da mesma aridade. Ou seja, precisam ter o mesmo número de atributos.
2. Os domínios do $i^{\underline{o}}$ atributo de r e do $i^{\underline{o}}$ atributo de s precisam ser o mesmo, para todo i.

Note que r e s podem ser relações de banco de dados ou relações temporárias que são o resultado das expressões de álgebra relacional.

A operação diferença de conjuntos

A operação **diferença de conjuntos**, indicada por –, permite encontrar tuplas que estejam em uma relação, mas não em outra. A expressão $r - s$ produz uma relação contendo as tuplas que estão em r mas não em s.

Podemos encontrar todos os clientes do banco que possuem uma conta mas não um empréstimo escrevendo

$$\Pi_{nome\_cliente}(depositante) - \Pi_{nome\_cliente}(tomador)$$

A relação resultante dessa consulta aparece na Figura 2.12.

Assim como a operação união, é preciso garantir que as diferenças de conjunto sejam tomadas entre relações compatíveis. Portanto, para uma operação diferença de conjuntos $r - s$ ser válida, exigimos que as relações r e s sejam da mesma aridade e que os domínios do $i^{\underline{o}}$ atributo de r e do $i^{\underline{o}}$ atributo de s sejam o mesmo.

A operação produto cartesiano

A operação **produto cartesiano**, indicada por um sinal de vezes (×), permite combinar informações de quaisquer duas relações. Escrevemos o produto cartesiano das relações r_1 e r_2 como $r_1 \times r_2$.

nome_cliente
Johnson
Lindsay
Turner

Figura 2.12 Clientes com uma conta, mas sem empréstimos.

Lembre-se de que uma relação é, por definição, um subconjunto de um produto cartesiano de um conjunto de domínios. A partir dessa definição, já devemos ter uma idéia sobre a definição da operação produto cartesiano, entretanto, como o mesmo nome de atributo pode aparecer tanto em r_1 quanto em r_2, deve-se criar um esquema de nomeação para distinguir entre esses atributos. Aqui, fazemos isso anexando a um atributo o nome da relação da qual o atributo originalmente veio. Por exemplo, o esquema de relação para $r = tomador \times empréstimo$ é

nome_cliente	*tomador.* *número_empréstimo*	*empréstimo.* *número_empréstimo*	*nome_agência*	*quantia*
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Adams	L-16	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Hayes	L-15	L-14	Downtown	1500
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Hayes	L-15	L-17	Downtown	1000
Hayes	L-15	L-23	Redwood	2000
Hayes	L-15	L-93	Mianus	500
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

Figura 2.13 Resultado de *tomador* × *empréstimo*.

(*tomador.nome_cliente, tomador.número_empréstimo, empréstimo.número_empréstimo, empréstimo.nome_agência, empréstimo.quantia*)

Com esse esquema, é possível distinguir *tomador.número_empréstimo* de *empréstimo.número_empréstimo*. Para os atributos que aparecem apenas em um dos dois esquemas, normalmente omite-se o prefixo *nome_relação*. Essa simplificação não gera qualquer ambigüidade. Podemos então escrever o esquema de relação para *r* como

(*nome_cliente, tomador.número_empréstimo, empréstimo.número_empréstimo, nome_agência, quantia*)

A convenção de nomeação exige que as relações que são os argumentos da operação produto cartesiano tenham nomes distintos. Essa exigência causa problemas em alguns casos, como quando o produto cartesiano de uma relação com ela mesma é desejado. Um problema semelhante surge se o resultado de uma expressão de álgebra relacional for utilizado em um produto cartesiano, já que precisaremos de um nome para a relação de modo que seja possível nos referir aos atributos da relação. Na seção "A operação renomeação", veremos como evitar esses problemas usando a operação renomeação.

Agora que conhecemos o esquema de relação para *r* = tomador × *empréstimo*, que tuplas aparecem em *r*? Como você pode suspeitar, construímos uma tupla de *r* de cada par de tuplas possível: uma da relação *tomador* e uma da relação *empréstimo*. Portanto, *r* é uma relação grande, como você pode ver pela Figura 2.13, que inclui apenas uma parte das tuplas que compõem *r*.

Considere que tenhamos n_1 tuplas em *tomador* e n_2 tuplas em *empréstimo*. Então, existem $n_1 * n_2$ maneiras de escolher um par de tuplas – uma tupla de cada relação; portanto, existem $n_1 * n_2$ tuplas em *r*. Em especial, observe que para algumas tuplas *t* em *r*, pode ser que $t[tomador.número\_empréstimo] \neq t[empréstimo.número\_empréstimo]$.

Em geral, se tivermos relações $r_1(R_1)$ e $r_2(R_2)$, então, $r_1 \times r_2$ é uma relação cujo esquema é a concatenação de R_1 e R_2. A relação *R* contém todas as tuplas *t* para as quais existe uma tupla t_1 em r_1 e uma tupla t_2 em r_2 para a qual $t[R_1] = t_1[R_1]$ e $t[R_2] = t_2[R_2]$.

Suponha que desejamos encontrar os nomes de todos os clientes que possuem um empréstimo na agência Perryridge. Para isso, são necessárias as informações na relação *empréstimo* e na relação *tomador*. Se escrevermos

$$\sigma_{nome\_agência = \text{"Perryridge"}}(tomador \times empréstimo)$$

então, o resultado é a relação na Figura 2.14. Temos uma relação que pertence apenas à agência Perryridge. Entretanto, a coluna *nome_cliente* pode conter clientes que não têm um empréstimo na agência Perryridge. (Se você não vir por que isso é verdade, lembre-se de que o produto cartesiano tira todos os pares possíveis de uma tupla de *tomador*

nome_cliente	*tomador. número_empréstimo*	*empréstimo. número_empréstimo*	*nome_agência*	*quantia*
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

Figura 2.14 Resultado de $\sigma_{nome\_agência = \text{"Perryridge"}}(tomador \times empréstimo)$.

com uma tupla de *empréstimo*.)

Como a operação produto cartesiano associa *toda* tupla de *empréstimo* com toda tupla de *tomador*, sabemos que, se um cliente tiver um empréstimo na agência Perryridge, então, existe alguma tupla em *tomador* × *empréstimo* que contenha seu nome, e *tomador.número_empréstimo* = *empréstimo.número_empréstimo*. Assim, se escrevermos

$$\sigma_{tomador.número\_empréstimo = empréstimo.número\_empréstimo} (\sigma_{nome\_agência = \text{"Perryridge"}} (tomador \times empréstimo))$$

obteremos apenas as tuplas de *tomador* × *empréstimo* que pertencem aos clientes que têm um empréstimo na agência Perryridge.

Finalmente, como queremos apenas *nome_cliente*, fazemos uma projeção:

$$\Pi_{nome\_cliente} (\sigma_{tomador.número\_empréstimo = empréstimo.número\_empréstimo} (\sigma_{nome\_agência = \text{"Perryridge"}} (tomador \times empréstimo)))$$

O resultado dessa expressão, mostrado na Figura 2.15, é a resposta correta à nossa consulta.

A operação renomeação

De modo diferente das relações no banco de dados, os resultados das expressões de álgebra relacional não possuem um nome para referenciá-las. É útil poder atribuir-lhes nomes; o operador **renomeação**, indicado pela letra grega minúscula ro (ρ), nos permite fazer isso. Considerando uma expressão de álgebra relacional E, a expressão

$$\rho_x (E)$$

retorna o resultado da expressão E sob o nome x.

Uma relação r por ela mesma é considerada uma expressão de álgebra relacional (trivial). Portanto, também é possível aplicar a operação renomeação a uma relação r para obter a mesma relação com um novo nome.

Uma segunda forma da operação renomeação é a seguinte. Considere que uma expressão de álgebra relacional E tenha aridade n. Então, a expressão

$$\rho_{x(A_1, A_2, \ldots, A_n)} (E)$$

retorna o resultado da expressão E com o nome x, e com os atributos renomeados para $A_1, A_2, \ldots, A_n$.

Para ilustrar a renomeação de uma relação, consideramos a consulta "Encontre o maior saldo no banco". Nossa estratégia é (1) calcular primeiro uma relação temporária consistindo nos saldos que *não* sejam o maior e (2) tomar a diferença de conjuntos entre a relação Π_{saldo} (*conta*) e a relação temporária recém-calculada, para obter o resultado.

Etapa 1: Para calcular a relação temporária, precisamos comparar os valores de todos os saldos de conta. Fazemos essa comparação calculando o produto cartesiano *conta* × *conta* e formando uma seleção para comparar o valor de quaisquer dois saldos aparecendo em uma tupla. Primeiro, precisamos criar um mecanismo para distinguir entre os dois atributos *saldo*. Usaremos a operação renomeação para renomear uma referência à relação *conta*; portanto, podemos referenciar a relação duas vezes sem ambigüidade.

Agora podemos escrever a relação temporária que consiste nos saldos que não são o maior:

$$\Pi_{conta.saldo} (\sigma_{conta.saldo < d.saldo} (conta \times \rho_d (conta)))$$

Essa expressão fornece os saldos na relação *conta* para os quais um saldo maior aparece em algum lugar na relação *conta* (renomeada como *d*). O resultado contém todos os saldos *exceto* o maior. A Figura 2.16 mostra essa relação.

Etapa 2: A consulta para encontrar o maior saldo no banco pode ser escrita como:

$$\Pi_{saldo} (conta) - \Pi_{conta.saldo} (\sigma_{conta.saldo < d.saldo} (conta \times \rho_d (conta)))$$

A Figura 2.17 mostra o resultado dessa consulta.

Como mais um exemplo da operação renomeação, considere a consulta "Encontre os nomes de todos os clientes que moram na mesma cidade e na mesma rua de Smith". Podemos obter a cidade e rua de Smith escrevendo

$$\Pi_{rua\_cliente,\ cidade\_cliente} (\sigma_{nome\_cliente = \text{"Smith"}} (cliente))$$

Entretanto, para encontrar outros clientes com essa cidade e rua, é preciso referenciar a relação *cliente* uma segunda vez. Na consulta a seguir, usamos a operação reno-

nome_cliente
Adams
Hayes

Figura 2.15 Resultado de $\Pi_{nome\_cliente} (\sigma_{tomador.número\_empréstimo = empréstimo.número\_empréstimo} (\sigma_{nome\_agência = \text{"Perryridge"}} (tomador \times empréstimo)))$.

Saldo
500
400
700
750
350

Figura 2.16 Resultado da subexpressão $\Pi_{conta.saldo}(\sigma_{conta.saldo < d.saldo}(conta \times \rho_d(conta)))$.

meação da expressão anterior para fornecer ao seu resultado o nome *endereço_smith* e para renomear seus atributos para *rua* e *cidade*, em vez de *rua_cliente* e *cidade_cliente*:

$$\Pi_{cliente.nome\_cliente}(\sigma_{cliente.rua\_cliente = endereço\_smith.rua \wedge cliente.cidade\_cliente = endereço\_smith.cidade}(cliente \times \rho_{endereço\_smith(rua,cidade)}(\Pi_{rua\_cliente,\ cidade\_cliente}(\sigma_{nome\_cliente = \text{"Smith"}}(cliente)))))$$

O resultado dessa consulta, quando aplicamos na relação *cliente* da Figura 2.4, aparece na Figura 2.18.

A operação renomeação não é estritamente necessária, já que é possível usar uma notação posicional para atributos. Podemos nomear atributos de uma relação implicitamente usando uma notação posicional, onde $1, $2,... refere-se ao primeiro atributo, ao segundo atributo e assim por diante. A notação posicional também se aplica aos resultados das operações de álgebra relacional. A seguinte expressão de álgebra relacional ilustra o uso da notação posicional com o operador unário σ:

$$\sigma_{\$2=\$3}(R \times R)$$

Se uma operação binária precisar distinguir entre suas duas relações de operando, uma notação posicional semelhante pode ser usada também para nomes de relação. Por exemplo, $R1 poderia se referir ao primeiro operando e $R2, ao segundo operando. Entretanto, a notação posicional é inconveniente para humanos, uma vez que a posição do atributo é um número, em vez de um nome de atributo fácil de lembrar. Conseqüentemente, não usamos a notação posicional neste livro.

Definição formal da álgebra relacional

As operações na seção "Operações fundamentais da álgebra relacional" nos permitem fornecer uma definição completa de uma expressão na álgebra relacional. Uma expressão básica na álgebra relacional consiste em um dos seguintes:

- Uma relação no banco de dados
- Uma relação constante

Uma relação constante é escrita listando suas tuplas dentro de { }, por exemplo, {(A-101, Downtown, 500) (A-215, Mianus, 700)}.

Uma expressão geral na álgebra relacional é construída a partir de subexpressões menores. Façamos com que E_1 e E_2 sejam expressões de álgebra relacional. Então, todas as expressões a seguir são expressões de álgebra relacional.

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$, onde P é um predicado em atributos em E_1
- $\Pi_S(E_1)$, onde S é uma lista consistindo em alguns dos atributos em E_1
- $\rho_x(E_1)$, onde x é o novo nome para o resultado de E_1

Outras operações de álgebra relacional

As operações fundamentais da álgebra relacional são suficientes para expressar qualquer consulta de álgebra relacional.[1] Entretanto, se nos restringirmos somente às opera-

[1] Na seção "Operações estendidas de álgebra relacional", apresentamos as operações que estendem a capacidade da álgebra relacional para manipular valores nulos e agregados.

Saldo
900

Figura 2.17 Maior saldo do banco.

nome_cliente
Curry
Smith

Figura 2.18 Clientes que moram na mesma cidade e na mesma rua de Smith.

ções fundamentais, certas consultas comuns serão extensas para expressar. Portanto, definimos as operações adicionais que não acrescentam qualquer capacidade à álgebra, mas simplificam consultas comuns. Para cada nova operação, fornecemos uma expressão equivalente que usa apenas as operações fundamentais.

A operação interseção de conjuntos

A primeira operação de álgebra relacional adicional que definiremos é a **interseção de conjuntos** ($\cap$). Suponha que desejamos encontrar todos os clientes que possuem um empréstimo e uma conta. Usando a interseção de conjuntos, podemos escrever

$$\Pi_{nome\_cliente}(tomador) \cap \Pi_{nome\_cliente}(depositante)$$

A relação resultante para essa consulta aparece na Figura 2.19.

Observe que podemos reescrever qualquer expressão de álgebra relacional que usa interseção de conjuntos substituindo a operação interseção por um par de operações de diferença de conjuntos, como:

$$r \cap s = r - (r - s)$$

Portanto, a interseção de conjuntos não é uma operação fundamental e não acrescenta qualquer capacidade à álgebra relacional. Apenas é mais conveniente escrever $r \cap s$ do que escrever $r - (r - s)$.

A operação junção natural

Muitas vezes é desejável simplificar certas consultas que exigem um produto cartesiano. Normalmente, uma consulta que envolve um produto cartesiano inclui uma operação de seleção no resultado do produto cartesiano. Considere a consulta "Encontre os nomes de todos os clientes que têm um empréstimo no banco, juntamente com o número e o valor do empréstimo". Primeiro formamos o produto cartesiano das relações *tomador* e *empréstimo*. Depois, selecionamos as tuplas que pertencem apenas ao mesmo *número_empréstimo*, seguido da projeção do *nome_cliente*, *número_empréstimo* e *quantia* resultantes:

$$\Pi_{nome\_cliente,\ empréstimo.número\_empréstimo,\ quantia}(\sigma_{tomador.número\_empréstimo\ =\ empréstimo.número\_empréstimo}(tomador \times empréstimo))$$

A *junção natural* é uma operação binária que permite combinar certas seleções e um produto cartesiano em uma única operação. Ela é indicada pelo símbolo de **junção** $\bowtie$. A operação junção natural forma um produto cartesiano dos seus dois argumentos, realiza uma seleção forçando igualdade nos atributos que aparecem nos dois esquemas de relação e, finalmente, remove atributos duplicados.

Embora a definição da junção natural seja complicada, a operação é fácil de aplicar. Como ilustração, considere novamente o exemplo "Encontre os nomes de todos os clientes que têm um empréstimo no banco e encontre a quantia do empréstimo". Expressamos essa consulta usando a junção natural desta forma:

$$\Pi_{nome\_cliente,\ número\_empréstimo,\ quantia}(tomador \bowtie empréstimo)$$

Como os esquemas para *tomador* e *empréstimo* (ou seja, *Esquema_tomador* e *Esquema_empréstimo*) possuem o atributo *número_empréstimo* em comum, a operação junção natural considera apenas pares de tuplas que têm o mesmo valor em *número_empréstimo*. Ela combina cada um desses pares de tuplas em uma única tupla na união dos dois esquemas (ou seja, *nome_cliente*, *nome_agência*, *número_em-*

nome_cliente
Hayes
Jones
Smith

Figura 2.19 Clientes com uma conta e um empréstimo no banco.

nome_cliente	número_empréstimo	quantia
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-23	2000
Smith	L-11	900
Williams	L-17	1000

Figura 2.20 Resultado de $\Pi_{nome\_cliente,\ número\_empréstimo,\ quantia}$ (*tomador* ⋈ *empréstimo*).

préstimo, quantia). Após realizar a projeção, obtemos a relação na Figura 2.20.

Considere dois esquemas de relação R e S – que são, é claro, listas de nomes de atributo. Se considerarmos os esquemas como sendo *conjuntos*, em vez de listas, podemos denotar os nomes de atributo que aparecem em R e S por $R \cap S$, e denotar os nomes de atributo que aparecem em R, em S ou em ambos por $R \cup S$. Da mesma forma, os nomes de atributo que aparecem em R mas não em S são indicados por $R - S$, enquanto $S - R$ denota os nomes de atributo que aparecem em S mas não em R. Observe que as operações união, interseção e diferença aqui estão em conjuntos de atributos, e não em relações.

Agora estamos prontos para uma definição formal da junção natural. Considere duas relações $r(R)$ e $s(S)$. A **junção natural de r e s**, indicada por $r \bowtie s$, é uma relação no esquema $R \cup S$ formalmente definida como:

$$r \bowtie s = \Pi_{R \cup S} (\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n} r \times s)$$

onde $R \cap S = \{A_1, A_2, \ldots, A_n\}$.

Como a junção natural é preponderante para muita da teoria e da prática de banco de dados relacional, daremos vários exemplos do seu uso.

- Encontre os nomes de todas as agências com clientes que têm uma conta no banco e que moram em Harrison.

$$\Pi_{nome\_agência} (\sigma_{cidade\_cliente = \text{"Harrison"}} (cliente \bowtie conta \bowtie depositante))$$

A relação resultante para essa consulta aparece na Figura 2.21.

Repare que escrevemos *cliente* ⋈ *conta* ⋈ *depositante* sem inserir parênteses para especificar a ordem em que as operações de junção natural nas três relações devem ser executadas. No caso anterior, existem duas possibilidades:

$$(cliente \bowtie conta) \bowtie depositante$$
$$cliente \bowtie (conta \bowtie depositante)$$

Não especificamos qual expressão pretendíamos porque as duas são equivalentes. Ou seja, a junção natural é **associativa**.

- Encontre todos os clientes que têm um empréstimo e uma conta no banco.

$$\Pi_{nome\_cliente} (tomador \bowtie depositante)$$

Note que na seção "A operação interseção de conjuntos" escrevemos uma expressão para essa consulta usando interseção de conjuntos. Repetimos essa expressão aqui:

$$\Pi_{nome\_cliente} (tomador) \cap \Pi_{nome\_cliente} (depositante)$$

A relação resultante para essa consulta apareceu anteriormente na Figura 2.19. Esse exemplo ilustra um fato geral sobre a álgebra relacional: É possível escrever várias expressões de álgebra relacional equivalentes que sejam bem diferentes uma das outras.

nome_agência
Brighton
Perryridge

Figura 2.21 Resultado de $\Pi_{nome\_agência}$ ($\sigma_{cidade\_cliente = \text{"Harrison"}}$ (*cliente* ⋈ *conta* ⋈ *depositante*)).

nome_agência
Brighton
Downtown

Figura 2.22 Resultado de $\Pi_{nome\_agência}$ ($\sigma_{cidade\_agência = \text{"Brooklyn"}}$ (**agência**)).

- Considere $r(R)$ e $s(S)$ relações sem quaisquer atributos em comum; isto é, $R \cap S = \varnothing$. ($\varnothing$ denota o conjunto vazio). Então, $r \bowtie s = r \times s$.

A operação *junção teta* é uma extensão da operação junção natural que permite combinar uma seleção e um produto cartesiano em uma única operação. Considere as relações $r(R)$ e $s(S)$ e seja θ um predicado em atributos no esquema $R \cup S$. A operação **junção teta** $r \bowtie_\theta s$ é definida desta forma:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

A operação divisão

A operação divisão, indicada por ÷, é apropriada para consultas que incluem a frase "para todo". Suponha que desejamos encontrar todos os clientes que têm uma conta em *todas* as agências localizadas em Brooklyn pela expressão

$$r_1 = \Pi_{nome\_agência}\ (\sigma_{cidade\_agência = \text{"Brooklyn"}}\ (agência))$$

A relação resultante para essa expressão aparece na Figura 2.22.

Podemos encontrar todos os pares (*nome_cliente, nome_agência*) para os quais o cliente tem uma conta em uma agência escrevendo

$$r_2 = \Pi_{nome\_cliente,\ nome\_agência}\ (depositante \bowtie conta)$$

A Figura 2.23 mostra a relação resultante para essa expressão.

Agora, precisamos encontrar clientes que aparecem em r_2 com *cada* nome de agência em r_1. A operação que fornece exatamente esses clientes é a divisão. Formulamos a consulta escrevendo

$$\Pi_{nome\_cliente,\ nome\_agência}\ (depositante \bowtie conta) \div \Pi_{nome\_agência}\ (\sigma_{cidade\_agência = \text{"Brooklyn"}}\ (agência))$$

O resultado dessa expressão é uma relação que tem o esquema (*nome_cliente*) e que contém a tupla (Johnson).

Formalmente, sejam $r(R)$ e $s(S)$ relações, e $S \subseteq R$; ou seja, cada atributo do esquema S também está no esquema R. A relação $r \div s$ é uma relação no esquema $R - S$ (isto é, no esquema contendo todos os atributos do esquema R que não estão no esquema S). Uma tupla t está em $r \div s$ se e somente se as duas condições forem satisfeitas:

1. t está em $\Pi_{R-S}(r)$
2. Para cada tupla t_s em s, existe uma tupla t_r em r satisfazendo estas duas condições:
   a. $t_r[S] = t_s[S]$
   b. $t_r[R - S] = t$

Talvez você se surpreenda ao descobrir que, dada uma operação de divisão e os esquemas das relações, podemos, na verdade, definir a operação divisão em função das operações fundamentais. Considere $r(R)$ e $s(S)$, com $S \subseteq R$:

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

Para verificar se esse exemplo é verdadeiro, observamos que $\Pi_{R-S}(r)$ fornece todas as tuplas t que satisfazem a pri-

nome_cliente	nome_agência
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

Figura 2.23 Resultado de $\Pi_{nome\_cliente,\ nome\_agência}$ (**depositante** $\bowtie$ **conta**).

meira condição da definição de divisão. A expressão no lado direito do operador de diferença de conjuntos

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

serve para eliminar as tuplas que não satisfazem a segunda condição da definição de divisão. Vejamos como ela faz isso. Considere $\Pi_{R-S}(r) \times s$. Essa relação está no esquema R e pareia cada tupla em $\Pi_{R-S}(r)$ com cada tupla em s. A expressão $\Pi_{R-S,S}(r)$ simplesmente reordena os atributos de r.

Assim, $(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$ nos fornece os pares de tuplas de $\Pi_{R-S}(r)$ e s que não aparecem em r. Se uma tupla t_j estiver em

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

então, existe alguma tupla t_i em s que não combina com a tupla t_j para formar uma tupla em r. Portanto, t_j contém um valor para atributos $R - S$ que não aparecem em $r \div s$. São esses valores que eliminamos de $\Pi_{R-S}(r)$.

A operação atribuição

Algumas vezes, é conveniente escrever uma expressão de álgebra relacional atribuindo partes dela a variáveis de relação temporárias. A operação **atribuição**, indicada por $\leftarrow$, age como a atribuição em uma linguagem de programação. Para ilustrar essa operação, considere a definição de divisão na seção "A operação divisão". Poderíamos escrever $r \div s$ como

$temp1 \leftarrow \Pi_{R-S}(r)$
$temp2 \leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r))$
$result = temp1 - temp2$

A avaliação de uma atribuição não resulta em qualquer relação sendo exibida para o usuário. Em vez disso, o resultado da expressão no lado direito do $\leftarrow$ é atribuído à variável de relação no lado esquerdo do $\leftarrow$. Essa variável de relação pode ser usada nas expressões subseqüentes.

Com a operação atribuição, uma consulta pode ser escrita como um programa seqüencial consistindo em uma série de atribuições seguida de uma expressão cujo valor é exibido como o resultado da consulta. Para consultas de álgebra relacional, a atribuição sempre precisa ser feita a uma variável de relação temporária. As atribuições a relações permanentes constituem uma modificação de banco de dados. Discutiremos esse aspecto na seção "Modificação do banco de dados". Note que a operação atribuição não fornece qualquer capacidade adicional para a álgebra. Entretanto, ela é uma maneira conveniente de expressar consultas complexas.

Operações estendidas de álgebra relacional

As operações básicas de álgebra relacional foram estendidas de várias formas. Uma extensão simples é permitir operações aritméticas como parte da projeção. Uma extensão importante é permitir *operações agregadas*, como calcular a soma dos elementos de um conjunto, ou sua média. Outra extensão importante é a operação *junção externa*, que permite que expressões de álgebra relacional lidem com valores nulos, que modelam informações ausentes.

Projeção generalizada

A operação **projeção generalizada** estende a operação projeção permitindo que funções aritméticas sejam usadas na lista de projeção. A operação projeção generalizada possui a forma

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

onde E é qualquer expressão de álgebra relacional e cada $F_1, F_2, \ldots, F_n$ é uma expressão aritmética envolvendo constantes e atributos no esquema de E. Como um caso especial, a expressão aritmética pode ser simplesmente um atributo ou uma constante.

Por exemplo, suponha que temos uma relação *info_crédito*, como na Figura 2.24, que lista o limite de crédito e as despesas até agora (o *saldo_crédito* na conta). Se quisermos saber quanto mais cada pessoa pode gastar, podemos escrever a seguinte expressão:

$$\Pi_{nome\_cliente,\ limite - saldo\_crédito}(info\_crédito)$$

nome_cliente	*limite*	*saldo_crédito*
Curry	2000	1750
Hayes	1500	1500
Jones	6000	700
Smith	2000	400

Figura 2.24 A relação *info_crédito*.

nome_cliente	crédito_disponível
Curry	250
Jones	5300
Smith	1600
Hayes	0

Figura 2.25 O resultado de $\Pi_{nome\_cliente,\ (limite - saldo\_crédito)\ \textbf{as}\ crédito\_disponível}$ (*info_crédito*).

O atributo resultante da expressão *limite – saldo_crédito* não possui um nome. Podemos aplicar a operação renomeação ao resultado da projeção generalizada para fornecer-lhe um nome. Como uma conveniência de notação, a renomeação de atributos pode ser combinada com projeção generalizada, como mostrado a seguir.

$$\Pi_{nome\_cliente,\ (limite - saldo\_crédito)\ \textbf{as}\ crédito\_disponível}(info\_crédito)$$

O segundo atributo dessa projeção generalizada recebeu o nome *crédito_disponível*. A Figura 2.25 mostra o resultado de aplicar essa expressão na relação da Figura 2.24.

Funções agregadas

As **funções agregadas** tomam uma coleção de valores e retornam um único valor como resultado. Por exemplo, a função agregada **sum** toma uma coleção de valores e retorna a soma dos valores. Portanto, a função **sum** aplicada na coleção

{1, 1, 3, 4, 4, 11}

retorna o valor 24. A função agregada **avg** retorna a média dos valores. Quando aplicada à coleção anterior, ela retorna o valor 4. A função agregada **count** retorna o número de elementos na coleção e retorna 6 na coleção anterior. Outras funções agregadas comuns incluem **min** e **max**, que retornam os valores mínimo e máximo em uma coleção; elas retornam 1 e 11, respectivamente, na coleção anterior.

As coleções em que as funções agregadas operam podem ter várias ocorrências de um valor; a ordem em que os valores aparecem não é relevante. Essas coleções são chamadas de **multiconjuntos**. Os conjuntos são um caso especial de multiconjuntos, em que há apenas uma cópia de cada elemento.

Para ilustrar o conceito de agregação, usaremos a relação *func_mp* da Figura 2.26, para funcionários de meio período. Suponha que queiramos encontrar a soma total dos salários de todos os funcionários de meio período no banco. A expressão de álgebra relacional para essa consulta é:

$$\mathcal{G}_{\text{sum}(salário)}(\text{func\_mp})$$

O símbolo $\mathcal{G}$ é a letra G em fonte caligráfica; leia-o como "G caligráfico". A operação de álgebra relacional G significa que agregação deve ser aplicada, e seu subscrito especifica a operação agregada a ser aplicada. O resultado dessa expressão é uma relação com um único atributo, contendo uma única linha com um valor numérico correspondente à soma de todos os salários de todos os funcionários trabalhando em meio período no banco.

$$\mathcal{G}_{\text{count-distinct}(nome\_agência)}(\text{func\_mp})$$

Para a relação na Figura 2.26, o resultado dessa consulta é uma única linha contendo o valor 3.

Suponha que queremos encontrar a soma total dos salários de todos os funcionários de meio período em cada

nome_funcionário	nome_agência	salário
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Rao	Austin	1500
Sato	Austin	1600

Figura 2.26 A relação *func_mp*.

nome_funcionário	*nome_agência*	*salário*
Rao	Austin	1500
Sato	Austin	1600
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300

Figura 2.27 A relação *func_mp* após o agrupamento.

agência do banco separadamente, em vez de a soma para o banco inteiro. Para isso, precisamos particionar a relação *func_mp* em **grupos** baseados na agência e aplicar a função agregada em cada grupo.

Há casos em que é necessário eliminar múltiplas ocorrências de um valor antes de calcular uma função agregada. Se quisermos eliminar duplicatas, usamos os mesmos nomes de função de antes, com a adição da string hifenizada "**distinct**" anexada no final do nome de função (por exemplo, **count-distinct**). Um exemplo pode ser visto na consulta "Encontre o número de agências aparecendo na relação *func_mp*". Nesse caso, um nome de agência é contado apenas uma vez, independente do número de funcionários trabalhando nessa agência. Escrevemos essa consulta da maneira a seguir. Esta expressão, que usa o operador de agregação $\mathcal{G}$, obtém o resultado desejado:

$$_{nome\_agência}\mathcal{G}_{\mathbf{sum}(salário)}(func\_mp)$$

Na expressão, o atributo *nome_agência* no subscrito do lado esquerdo de $\mathcal{G}$ indica que a relação de entrada *func_mp* precisa ser dividida em grupos com base no valor de *nome_agência*. A Figura 2.27 mostra os grupos resultantes. A expressão **sum**(*salário*) no subscrito do lado direito de $\mathcal{G}$ indica que para cada grupo de tuplas (ou seja, cada agência), a função de agregação **sum** precisa ser aplicada na coleção de valores do atributo *salário*. A relação de saída consiste em tuplas com o nome da agência e a soma dos salários para a agência, como mostra a Figura 2.28.

A forma geral da **operação agregação** $\mathcal{G}$ é a seguinte:

$$G_{1, G_2, \ldots, G_n}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots, F_m(A_m)}(E)$$

onde E é qualquer expressão de álgebra relacional; G_1, G_2,..., G_n constitui uma lista de atributos em que agrupar; cada F_i é uma função agregada; cada A_i é um nome de atributo. O significado da operação é o que segue. As tuplas no resultado da expressão E são particionadas em grupos de tal maneira que

1. Todas as tuplas em um grupo têm os mesmos valores para $G_1, G_2, \ldots, G_n$.
2. Tuplas em grupos diferentes têm valores diferente para $G_1, G_2, \ldots, G_n$.

Portanto, os grupos podem ser identificados pelos valores dos atributos $G_1, G_2, \ldots, G_n$. Para cada grupo $(g_1, g_2, \ldots, g_n)$, o resultado tem uma tupla $(g_1, g_2, \ldots, g_n, a_1, a_2, \ldots, a_m)$ onde, para cada i, a_i é o resultado de aplicar a função agregada F_i no multiconjunto de valores para o atributo A_i no grupo.

Como um caso especial da operação agregada, a lista de atributos $G_1, G_2, \ldots, G_n$ pode ser vazia, caso em que há um único grupo contendo todas as tuplas na relação. Isso corresponde à agregação sem agrupamento.

Voltando ao nosso exemplo anterior, se quisermos encontrar o maior salário de funcionários de meio período, além da soma dos salários, escrevemos a expressão

$$_{nome\_agência}\mathcal{G}_{\mathbf{sum}(salário),\ \mathbf{max}(salário)}(func\_mp)$$

nome_agência	*sum(salário)*
Austin	3100
Downtown	5300
Perryridge	8100

Figura 2.28 Resultado de $_{nome\_agência}\mathcal{G}_{\mathbf{sum}(salário)}(func\_mp)$.

nome_agência	sum_salário	max_salário
Austin	3100	1600
Downtown	5300	2500
Perryridge	8100	5300

Figura 2.29 Resultado de $_{nome\_agência}\mathcal{G}_{\mathbf{sum}(salário)\ \mathbf{as}\ soma\_salário,\ \mathbf{max}(salário)\ \mathbf{as}\ max\_salário}(func\_mp)$.

Como na projeção generalizada, o resultado de uma operação de agregação não possui um nome. Podemos aplicar uma operação renomeação ao resultado para fornecer-lhe um nome. Como uma conveniência notacional, os atributos de uma operação de agregação podem ser renomeados como ilustrado a seguir.

$$_{nome\_agência}\mathcal{G}_{\mathbf{sum}(salário)\ \mathbf{as}\ soma\_salário,\ \mathbf{max}(salário)\ \mathbf{as}\ max\_salário}(func\_mp)$$

A Figura 2.29 mostra o resultado da expressão.

Junção externa

A operação **junção externa** é uma extensão da operação junção para lidar com informações mescladas. Suponha que temos as relações com os seguintes esquemas, que contêm dados sobre funcionários de período integral:

funcionário(*nome_funcionário, rua,* cidade)
func_pi (*nome_funcionário, nome_agência, salário*)

Considere as relações *funcionário* e *func_pi* na Figura 2.30. Suponha que queremos gerar uma única relação com todas as informações (rua, cidade, nome da agência e salário) sobre os funcionários de período integral. Um método possível seria usar a operação junção natural desta forma:

$$funcionário \bowtie func\_pi$$

O resultado dessa expressão aparece na Figura 2.31. Observe que perdemos as informações sobre rua e cidade de Smith, já que a tupla descrevendo Smith está ausente da relação *func_pi*; da mesma forma, perdemos as informações sobre nome de agência e salário de Gates, já que a tupla descrevendo Gates está ausente da relação *funcionário*.

Podemos usar a operação *junção externa* para evitar essa perda de informações. Na verdade, existem três formas da operação: *junção externa esquerda*, indicada por $\mathbin{⟕}$; *junção externa direita*, indicada por $\mathbin{⟖}$; e *junção externa completa*, indicada por $\mathbin{⟗}$. Todas as três formas de junção externa calculam a junção e acrescentam tuplas extras ao resultado da junção. Os resultados das expressões *funcionário* $\mathbin{⟕}$ *func_pi*, *funcionário* $\mathbin{⟖}$ *func_pi* e *funcionário* $\mathbin{⟗}$ *func_pi* aparecem nas Figuras 2.32, 2.33 e 2.34, respectivamente.

A **junção externa esquerda** ($\mathbin{⟕}$) toma todas as tuplas na relação esquerda que não correspondem à tupla alguma na relação direita, preenche as tuplas com valores nulos para todos os outros atributos da relação direita e os acrescenta ao resultado da junção natural. Na Figura 2.32, a tupla (Smith, Revolver, Death Valley, *null*, *null*) é uma dessas tuplas. Todas as informações da relação esquerda estão presentes no resultado da junção externa esquerda.

nome_funcionário	rua	Cidade
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

nome_funcionário	nome_agência	salário
Coyote	Mesa	1500
Rabbit	Mesa	1300
Smith	Redmond	5300
Williams	Redmond	1500

Figura 2.30 Relações *funcionário* e *func_pi*.

nome_funcionário	Rua	cidade	nome_agência	salário
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500

Figura 2.31 O resultado de *funcionário* ⋈ *func_pi*.

A **junção externa direita** (⟖) é simétrica à junção externa esquerda: Ela preenche com nulos as tuplas da relação direita que não correspondem a tupla alguma da relação esquerda e as acrescenta ao resultado da junção natural. Na Figura 2.33, a tupla (Gates, *nulo*, *nulo*, Redmond, 5300) é uma dessas tuplas. Portanto, todas as informações da relação direita estão presentes no resultado da junção externa direita.

A **junção externa completa** (⟗) realiza as duas operações, preenchendo as tuplas da relação esquerda que não correspondem a tupla alguma da relação direita, bem como as tuplas da relação direita que não correspondem a tupla alguma da relação esquerda, e acrescentando-as ao resultado da junção. A Figura 2.34 mostra o resultado de uma junção externa completa.

Como as operações de junção externa podem gerar resultados contendo valores nulos, precisamos especificar como as diferentes operações de álgebra relacional lidam com valores nulos. A próxima seção trata dessa questão.

É interessante notar que as operações de junção externa podem ser expressas pelas operações básicas de álgebra relacional. Por exemplo, a operação de junção externa esquerda, $r ⟕ s$, pode ser escrita como

$$(r \bowtie s) \cup (r - \Pi_R(r \bowtie s)) \times \{(null, \ldots, null)\}$$

onde a relação constante $\{(null, \ldots, null)\}$ está no esquema $S - R$.

Valores nulos

Nesta seção, definiremos como as várias operações de álgebra relacional lidam com valores nulos e as complicações que surgem quando há um valor nulo em uma operação aritmética ou em uma comparação. Como veremos, normalmente há mais de uma maneira possível de lidar com valores nulos, e, consequentemente, nossas definições algumas vezes podem ser arbitrárias. Portanto, as operações e comparações em valores nulos devem ser evitadas, onde possível.

Como o valor especial *nulo* indica "valor desconhecido ou inexistente", quaisquer operações aritméticas (como +, −, *, /) envolvendo valores nulos precisam retornar um resultado nulo.

Da mesma forma, quaisquer comparações (como <, <=, >, >=, ≠) envolvendo um valor nulo são avaliadas para o valor especial **desconhecido**; não podemos dizer ao certo se o resultado da comparação é verdadeiro ou falso; portanto, dizemos que o resultado é o novo valor de verdade *desconhecido*.

nome_funcionário	Rua	cidade	nome_agência	salário
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*nulo*	*nulo*

Figura 2.32 Resultado de *funcionário* ⟕ *func_pi*.

nome_funcionário	Rua	cidade	nome_agência	salário
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	*Nulo*	*nulo*	Redmond	5300

Figura 2.33 Resultado de *funcionário* ⟖ *func_pi*.

nome_funcionário	Rua	cidade	nome_agência	salário
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*nulo*	*nulo*
Gates	*Nulo*	*nulo*	Redmond	5300

Figura 2.34 Resultado de *funcionário* ⟕ *func_pi*.

As comparações envolvendo nulos podem ocorrer dentro de expressões booleanas envolvendo as operações **e**, **ou** e **não**. Portanto, precisamos definir como as três operações booleanas lidam com o valor de verdade *desconhecido*.

- **e**: (*verdadeiro* **e** *desconhecido*) = desconhecido; (*falso* e desconhecido) = *falso*; (*desconhecido* e *desconhecido*) = *desconhecido*.
- **or**: (*verdadeiro* **ou** *desconhecido*) = verdadeiro; (*falso* ou desconhecido) = *desconhecido*; (*desconhecido* ou *desconhecido*) = *desconhecido*.
- **não**: (**não** *desconhecido*) = *desconhecido*.

Agora estamos prontos para descrever como as diferentes operações relacionais lidam com valores nulos.

- **Seleção**: a operação seleção avalia o predicado P em $\sigma_P(E)$ em cada tupla t em E. Se o predicado retornar o valor *verdadeiro*, t é acrescentado ao resultado. Caso contrário, se o predicado retornar *desconhecido* ou *falso*, t não é acrescentado ao resultado.
- **Junção**: as junções podem ser expressas como um produto cartesiano seguido de uma seleção. Portanto, a definição de como a seleção manipula nulos também estabelece como as operações de junção manipulam nulos.

  Em uma junção natural, digamos, $r \bowtie s$, podemos ver que se duas tuplas, $t_r \in r$ e $t_s \in s$, possuem ambas um valor nulo em um atributo comum, então as tuplas não serão correspondentes.
- **Projeção**: a operação projeção trata nulos exatamente como qualquer outro valor ao eliminar duplicatas. Assim, se duas tuplas no resultado da projeção forem exatamente iguais, e ambas possuírem nulos nos mesmos campos, elas são tratadas como duplicatas.

  A decisão é um tanto arbitrária, já que, sem conhecer o valor real, não sabemos se as duas instâncias do nulo são duplicatas ou não.
- **União, interseção, diferença**: essas operações tratam nulos exatamente como a operação projeção; elas tratam as tuplas que têm os mesmos valores em todos os campos como duplicatas, mesmo se alguns dos campos tiverem valores nulos em ambas as tuplas.

  O comportamento é bastante arbitrário, especialmente nos casos da interseção e da diferença, já que não sabemos se os valores reais (se houver) representados pelos nulos são iguais.
- **Projeção generalizada**: descrevemos como os nulos são manipulados nas expressões no início desta seção. Tuplas duplicatas contendo valores nulos são manipulados como na operação projeção.
- **Agregação**: quando nulos ocorrem em atributos de agrupamento, a operação agregada os trata exatamente como na projeção: Se duas tuplas são iguais em todos os atributos de agrupamento, a operação as coloca no mesmo grupo, mesmo se alguns de seus valores de atributo forem nulos.

  Quando nulos ocorrem em atributos agregados, a operação exclui os valores nulos no início, antes de aplicar a agregação. Se o multiconjunto resultante for vazio, o resultado agregado será nulo.

  Observe que o tratamento de nulos aqui é diferente do tratamento nas expressões aritméticas comuns; poderíamos ter definido o resultado de uma operação agregada como nulo mesmo se um dos valores agregados fosse nulo. Entretanto, isso significaria que um único valor desconhecido em um grupo grande faria com que o resultado agregado no grupo fosse nulo, e perderíamos muita informação útil.
- **Junção externa**: as operações de junção externa se comportam exatamente como operações de junção natural, exceto nas tuplas que não ocorrem no resultado da junção. Essas tuplas podem ser acrescentadas ao resultado (dependendo de se a operação é ⟕, ⟖ ou ⟗), preenchidas com nulos.

Modificação do banco de dados

Até agora, limitamos nossa atenção à extração de informações do banco de dados. Nesta seção, analisaremos como incluir, remover ou alterar informações no banco de dados.

Expressamos modificações de banco de dados usando a operação atribuição. Fazemos atribuição nas relações de banco de dados reais usando a mesma notação que a descrita na seção "Outras operações de álgebra relacional" para atribuição.

Exclusão

Expressamos uma requisição de exclusão de maneira quase idêntica a uma consulta. Entretanto, em vez de exibir tuplas ao usuário, removemos as tuplas selecionadas do banco de dados. É possível excluir apenas tuplas inteiras, e não valores em atributos específicos. Na álgebra relacional, uma exclusão é expressa por

$$r \leftarrow r - E$$

onde r é uma relação e E é uma consulta de álgebra relacional.

Aqui estão alguns exemplos de requisições de exclusão de álgebra relacional:

- Exclua todos os registros de conta de Smith.

$$depositante \leftarrow depositante - \sigma_{nome\_cliente = \text{"Smith"}}(depositante)$$

- Exclua todos os empréstimos com quantia na faixa de 0 a 50.

$$empréstimo \leftarrow empréstimo - \sigma_{quantia \geq 0 \wedge quantia \leq 50}(empréstimo)$$

- Exclua todas as contas em agências localizadas no Brooklyn.

$$r_1 \leftarrow \sigma_{cidade\_agência = \text{"Brooklyn"}}(conta \bowtie agência)$$
$$r_2 \leftarrow \Pi_{nome\_agência,\ número\_conta,\ saldo}(r_1)$$
$$conta \leftarrow conta - r_2$$

Observe que, no último exemplo, simplificamos a expressão usando atribuição a relações temporárias (r_1 e r_2).

Inserção

Para inserir dados em uma relação, especificamos uma tupla a ser inserida ou escrevemos uma consulta cujo resultado seja um conjunto de tuplas a ser inserido. Obviamente, os valores de atributo para tuplas inseridas precisam ser membros do domínio do atributo. Da mesma forma, as tuplas a serem inseridas precisam ser da aridade correta. A álgebra relacional expressa uma inserção por

$$r \leftarrow r \cup E$$

onde r é uma relação e E é uma expressão de álgebra relacional. Expressamos a inserção de uma única tupla fazendo E ser uma relação constante contendo uma tupla.

Suponha que desejamos inserir o fato de que Smith tem $1.200 na conta A-973 na agência Perryridge. Escrevemos

$$conta \leftarrow conta \cup \{(\text{A-973}, \text{"Perryridge"}, 1200)\}$$
$$depositante \leftarrow depositante \cup \{(\text{"Smith"}, \text{A-973})\}$$

De modo ainda mais amplo, poderíamos querer inserir tuplas com base no resultado de uma consulta. Suponha que queremos fornecer, como um presente para todos os clientes de empréstimo da agência Perryridge, uma nova conta de poupança de $200. O número de empréstimo nos servirá como o número de conta para essa conta de poupança. Escrevemos

$$r_1 \leftarrow (\sigma_{nome\_agência = \text{"Perryridge"}}(tomador \bowtie empréstimo))$$
$$r_2 \leftarrow \Pi_{número\_empréstimo,\ nome\_agência}(r_1)\ conta \leftarrow$$
$$conta \cup (r_2 \times \{(200)\})\ depositante \leftarrow$$
$$depositante \cup \Pi_{nome\_cliente,\ número\_empréstimo}(r_1)$$

Em vez de especificar uma tupla como fizemos anteriormente, especificamos um conjunto de tuplas, que é inserido nas relações *conta* e *depositante*. Cada tupla na relação *conta* tem um *número_conta* (que é o mesmo *número_empréstimo*), um *nome_agência* (Perryridge) e o saldo inicial da nova conta ($200). Cada tupla na relação *depositante* possui como *nome_cliente* o nome do cliente do empréstimo que está recebendo a nova conta e o mesmo número de conta da tupla *conta* correspondente.

Atualização

Em certas situações, podemos querer mudar um valor em uma tupla sem mudar *todos* os valores na tupla. Podemos usar o operador de projeção generalizada para realizar essa tarefa:

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(r)$$

onde cada F_i é o i^o atributo de r, se o i^o atributo não estiver atualizado, ou, se o atributo precisar ser atualizado, F_i é uma expressão, envolvendo apenas constantes e os atributos de r, que fornece o novo valor para o atributo. Note que o esquema da expressão resultante da projeção generalizada precisa corresponder ao esquema original de r.

Se quisermos selecionar algumas tuplas de r para atualizar apenas elas, podemos usar a expressão a seguir; aqui, P denota a condição de seleção que escolhe quais tuplas atualizar:

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(\sigma_P(r)) \cup (r - \sigma_P(r))$$

Para ilustrar o uso da operação de atualização, suponha que pagamentos de juros estão sendo feitos e que todos os saldos devem ser acrescidos de 5%. Escrevemos

$$conta \leftarrow \Pi_{número\_conta,\ nome\_agência,\ saldo\ *\ 1,05}(conta)$$

Agora suponha que as contas com saldos acima de $10.000 recebem 6% de juros, enquanto todas as outras recebem 5%. Escrevemos

$$conta \leftarrow \Pi_{número\_conta,\ nome\_agência,\ saldo\ *\ 1,06}(\sigma_{saldo\ >\ 10000}(conta)) \cup \Pi_{número\_conta,\ nome\_agência,\ saldo\ *\ 1,05}(\sigma_{saldo\ \leq\ 10000}(conta))$$

Resumo

- O **modelo de dados relacional** é baseado em uma coleção de tabelas. O usuário do sistema de banco de dados pode consultar essas tabelas, inserir novas tuplas, excluir tuplas e atualizar (modificar) tuplas. Há várias linguagens para expressar essas operações.
- A **álgebra relacional** define um conjunto de operações algébricas que atuam sobre as tabelas e geram tabelas como suas saídas. Essas operações podem ser combinadas para obter expressões que denotam as consultas desejadas. A álgebra define as operações básicas usadas dentro das linguagens de consulta relacional.
- As operações na álgebra relacional podem ser divididas em
  - Operações básicas
  - Operações adicionais que podem ser expressas em termos das operações básicas
  - Operações estendidas, algumas das quais acrescentam mais capacidade expressiva à álgebra relacional
- Os bancos de dados podem ser modificados por **inserção**, **exclusão** ou **atualização** de tuplas. Usamos a álgebra relacional com o **operador de atribuição** para expressar essas modificações.
- A álgebra relacional é uma linguagem concisa e formal que é imprópria para usuários casuais de um sistema de banco de dados. Os sistemas de banco de dados comerciais, portanto, usam linguagens com mais "açúcar sintático". Nos Capítulos 3 e 4, consideraremos a linguagem mais influente – a SQL, que é baseada na álgebra relacional.

Termos de revisão

- Tabela
- Relação
- Variável de tupla
- Domínio atômico
- Valor nulo
- Esquema de banco de dados
- Instância de banco de dados
- Esquema de relação
- Instância de relação
- Chaves
- Chave estrangeira
  - Relação referenciadora
  - Relação referenciada
- Diagrama de esquema
- Linguagem de consulta
- Linguagem procedural
- Linguagem não procedural
- Álgebra relacional
- Operações de álgebra relacional
  - Seleção σ
  - Projeção Π
  - União $\cup$
  - Diferença de conjuntos $-$
  - Produto cartesiano $\times$
  - Renomeação ρ
- Operações adicionais
  - Interseção de conjuntos $\cap$
  - Junção natural $\bowtie$
  - Divisão $\div$
- Operação de atribuição
- Operações de álgebra relacional estendidas
  - Projeção generalizada Π
  - Junção externa
    - Junção externa esquerda $\mathbin{⟕}$
    - Junção externa direita $\mathbin{⟖}$
    - Junção externa completa $\mathbin{⟗}$
  - Agregação $\mathcal{G}$
- Multiconjuntos
- Agrupamento
- Valor nulo
- Valores de verdade
  - *verdadeiro*
  - *falso*
  - *desconhecido*
- Modificação do banco de dados
  - Exclusão
  - Inserção
  - Atualização

Exercícios práticos

2.1 Considere o banco de dados relacional da Figura 2.35, no qual as chaves primárias estão sublinhadas. Forneça uma expressão na álgebra relacional para expressar cada uma das seguintes consultas:

a. Encontre os nomes de todos os funcionários que moram na mesma cidade e na mesma rua de seus gerentes.

b. Encontre os nomes de todos os funcionários nesse banco de dados que não trabalham para o First Bank Corporation.
c. Encontre os nomes de todos os funcionários que ganham mais do que todos os funcionários do Small Bank Corporation.

2.2 As operações de junção externa estendem a operação de junção natural para que as tuplas das relações participantes não sejam perdidas no resultado da junção. Descreva como a operação de junção teta pode ser estendida para que as tuplas das relações esquerda, direita ou ambas não sejam perdidas do resultado de uma junção teta.

2.3 Considere o banco de dados relacional da Figura 2.35. Dê uma expressão na álgebra relacional para cada requisição:
a. Modifique o banco de dados de modo que Jones agora more em Newtown.
b. Dê um aumento de 10% no salário de todos os gerentes nesse banco de dados.

Exercícios

2.4 Descreva as diferenças de significado entre os termos *relação* e *esquema de relação*.

2.5 Considere o banco de dados relacional da Figura 2.35, no qual as chaves primárias são sublinhadas. Forneça uma expressão na álgebra relacional para expressar cada uma das seguintes consultas:
a. Encontre os nomes de todos os funcionários que trabalham para o First Bank Corporation.
b. Encontre os nomes e cidades de residência de todos os funcionários que trabalham para o First Bank Corporation.
c. Encontre os nomes, ruas e cidades de todos os funcionários que trabalham para o First Bank Corporation e ganham mais de $10.000 por ano.
d. Encontre os nomes de todos os funcionários nesse banco de dados que moram na mesma cidade da empresa para a qual trabalham.
e. Considerando que as empresas podem estar localizadas em várias cidades, encontre todas as empresas em cada cidade onde a Small Bank Corporation está localizada.

2.6 Considere a relação da Figura 2.20, que mostra o resultado da consulta "Encontre os nomes de todos os clientes que têm um empréstimo no banco". Reescreva a consulta para incluir não apenas o nome, mas também a cidade onde mora cada cliente. Observe que, agora, o cliente Jackson não aparece mais no resultado, ainda que ele realmente tenha um empréstimo do banco.
a. Explique por que Jackson não aparece no resultado.
b. Suponha que você queira que Jackson apareça no resultado. Como modificaria o banco de dados para conseguir esse intento?
c. Novamente, suponha que você queira que Jackson apareça no resultado. Escreva uma consulta usando uma junção externa que realize essa tarefa sem precisar modificar o banco de dados.

2.7 Considere o banco de dados relacional da Figura 2.35. Forneça uma expressão na álgebra relacional para cada requisição:
a. Dê um aumento salarial de 10% a todos os funcionários do First Bank Corporation.
b. Dê um aumento salarial de 10% a todos os gerentes do First Bank Corporation, a menos que o salário seja maior que $100.000. Nesses casos, dê um aumento de apenas 3%.
c. Exclua todas as tuplas na relação *trabalha* para funcionários do Small Bank Corporation.

2.8 Usando o exemplo de banco, escreva consultas de álgebra relacional para encontrar as contas mantidas por mais de dois clientes das seguintes maneiras:
a. Usando uma função agregada.
b. Sem usar quaisquer funções agregadas.

2.9 Considere o banco de dados relacional da Figura 2.35. Dê uma expressão de álgebra relacional para cada uma das seguintes consultas:
a. Encontre a empresa que possui mais funcionários.
b. Encontre a empresa com a menor folha de pagamento.

funcionário (<u>*nome_pessoa*</u>, *rua*, *cidade*)
trabalha (<u>*nome_pessoa*</u>, *nome_empresa*, *salário*)
empresa (<u>*nome_empresa*</u>, *cidade*)
gerencia (<u>*nome_pessoa*</u>, *nome_gerente*)

Figura 2.35 Banco de dados relacional para os Exercícios 2.1, 2.3, 2.5, 2.7 e 2.9.

c. Encontre as empresas cujos funcionários ganham um salário médio maior do que o salário médio no First Bank Corporation.

2.10 Cite duas razões por que valores nulos poderiam ser introduzidos no banco de dados.

2.11 Considere o seguinte esquema relacional:

funcionário(*num_emp*, *nome*, *escritório*, *idade*)
livros (*isbn*, *título*, *autores*, *editora*)
empréstimo (*num_emp*, *isbn*, *data*)

Escreva as seguintes consultas na álgebra relacional.

a. Encontre os nomes dos funcionários que tomaram emprestado um livro publicado pela Elsevier.
b. Encontre os nomes dos funcionários que tomaram emprestados todos os livros publicados pela Elsevier.
c. Encontre os nomes dos funcionários que tomaram emprestados mais de cinco livros diferentes publicados pela Elsevier.
d. Para cada editora, encontre os nomes dos funcionários que tomaram emprestados mais de cinco livros dessa editora.

Notas bibliográficas

E. F. Codd, do IBM San Jose Research Laboratory, propôs o modelo relacional no final da década de 1960 (Codd [1970]). Este trabalho rendeu a Codd o prestigiado prêmio ACM Turing Award em 1981 (Codd [1982]).

Após Codd ter publicado seu ensaio original, vários projetos de pesquisa foram formados com o objetivo de construir sistemas de banco de dados relacionais incluindo o System R no San Jose Research Laboratory, o Ingres na Universidade da Califórnia em Berkeley e o Query-by-Example no IBM T. J. Watson Research Center.

Atzeni e Antonellis [1993] e Maier [1983] são textos dedicados exclusivamente à teoria do modelo de dados relacional.

A definição original de álgebra relacional está em Codd [1970]. Extensões ao modelo relacional e discussões da incorporação de valores nulos na álgebra relacional (o modelo RM/T), bem como junções externas, estão em Codd [1979]. Codd [1990] é um compêndio dos ensaios de E. F. Codd sobre o modelo relacional. As junções externas também são abordadas em Date [1993b].

Muitos produtos de banco de dados relacional estão agora comercialmente disponíveis. Estes incluem o IBM DB2, Oracle, Sybase, Informix e Microsoft SQL Server. Os sistemas de banco de dados relacionais de fonte aberta incluem o MySQL e o PostgreSQL. Os produtos de banco de dados projetados para uso pessoal incluem o Microsoft Access e o FoxPro.

CAPÍTULO 3

SQL

A álgebra relacional descrita no Capítulo 2 fornece uma notação formal e concisa para representar consultas. Entretanto, os sistemas de banco de dados comerciais exigem uma linguagem de consulta que seja mais amigável. Neste capítulo, bem como no Capítulo 4, estudaremos a SQL, a linguagem de consulta mais comercialmente influente. A SQL usa uma combinação de construções de álgebra relacional (Capítulo 2) e cálculo relacional (Capítulo 5).

Embora nos referimos à SQL como uma "linguagem de consulta", ela pode fazer muito mais do que simplesmente consultar um banco de dados. Ela pode definir a estrutura dos dados, modificar dados no banco de dados e especificar restrições de segurança.

Não é nossa intenção fornecer um guia do usuário completo para a SQL. Em vez disso, apresentamos as construções e conceitos fundamentais da SQL. Implementações individuais da SQL podem diferir nos detalhes ou aceitar apenas um subconjunto da linguagem completa.

História

A IBM desenvolveu a versão original da SQL, originalmente chamada Sequel, como parte do projeto R no início da década de 1970. A linguagem Sequel evoluiu desde então, e seu nome mudou para SQL (Structured Query Language). Muitos produtos agora aceitam a linguagem SQL. A SQL se estabeleceu claramente como a linguagem padrão de banco de dados relacional.

Em 1986, o American National Standards Institute (ANSI) e a International Organization for Standardization (ISO) publicaram um padrão SQL, chamado SQL-86. Em 1989, o ANSI publicou um padrão estendido para a linguagem: a SQL-89. A próxima versão do padrão foi a SQL-92, seguida da SQL:1999; a versão mais recente é a SQL:2003. As notas bibliográficas fornecem referências para esses padrões.

A linguagem SQL possui várias partes:

- **Linguagem de definição de dados** (DDL). A DDL da SQL fornece comandos para definir esquemas de relação, excluir relações e modificar esquemas.
- **Linguagem de manipulação de dados** interativa (DML). A DML da SQL inclui uma linguagem de consulta baseada na álgebra relacional (2) e no cálculo relacional de tupla (5). Ela também inclui comandos para inserir, excluir e modificar tuplas no banco de dados.
- **Integridade**. A DDL SQL inclui comandos para especificar restrições de integridade às quais os dados armazenados no banco de dados precisam satisfazer. As atualizações que violam as restrições de integridade são proibidas.
- **Definição de view**. A DDL SQL inclui comandos para definir views.
- **Controle de transação**. A SQL inclui comandos para especificar o início e o fim das transações.
- **SQL embutida e SQL dinâmica**. A SQL embutida e a dinâmica definem como as instruções SQL podem ser incorporadas dentro das linguagens de programação de finalidade geral, como C, C++, Java, PL/I, Cobol, Pascal e Fortran.
- **Autorização**. A DDL SQL inclui comandos para especificar direitos de acesso para relações e views.

Neste capítulo, apresentamos uma análise dos recursos básicos de DML e DDL da SQL. Nossa descrição se baseia ainda mais no padrão amplamente implementado, a SQL-92.

mas também abordamos algumas extensões dos padrões SQL:1999 e SQL:2003.

No Capítulo 4 fornecemos um estudo mais detalhado do sistema de tipo, restrições de integridade e autorização da SQL. Nesse capítulo, também descrevemos brevemente a SQL embutida e dinâmica, incluindo os padrões ODBC e JDBC para interagir com um banco de dados de programas escritos nas linguagens C e Java. No Capítulo 9, resumimos as extensões orientadas a objeto da SQL que foram introduzidas na SQL:1999.

Muitos sistemas de banco de dados aceitam a maioria do padrão SQL-92 e algumas das novas construções do SQL:1999 e SQL:2003, embora atualmente nenhum sistema de banco de dados aceite todas as novas construções. Você também deve estar ciente de que muitos sistemas de banco de dados não aceitam alguns recursos da SQL-92 e que muitos bancos de dados fornecem recursos não padronizados não abordados aqui. Caso descubra que alguns recursos de linguagem descritos aqui não funcionam no sistema de banco de dados que você usa, consulte os manuais do usuário para seu sistema de banco de dados a fim de saber exatamente quais recursos ele aceita.

A empresa que usamos nos exemplos deste capítulo e dos seguintes é uma instituição bancária. A Figura 3.1 fornece o esquema relacional que usamos em nossos exemplos, com os atributos de chave primária sublinhados. Lembre-se de que, no Capítulo 2, primeiro definimos um esquema de relação R listando seus atributos e, depois, definimos uma relação r no esquema usando a notação $r(R)$. A notação na Figura 3.1 omite o nome do esquema e define o esquema de uma relação listando diretamente seus atributos.

Definição de dados

O conjunto de relações em um banco de dados precisa ser especificado para o sistema por meio de uma linguagem de definição de dados (DDL). A DDL SQL permite a especificação não só de um conjunto de relações, mas também de informações sobre cada relação, incluindo

- O esquema para cada relação
- O domínio dos valores associados a cada atributo
- As restrições de integridade
- O conjunto dos índices a serem mantidos para cada relação
- As informações de segurança e autorização para cada relação
- A estrutura de armazenamento físico de cada relação no disco

Discutimos aqui a definição de esquema básica e os valores de domínio básicos, deixando para o Capítulo 4 a análise dos outros recursos de DDL da SQL.

Tipos de domínio básicos

O padrão SQL aceita diversos tipos de domínio internos, incluindo:

- **char**(n): uma string de caracteres de tamanho fixo com tamanho n especificado pelo usuário. A forma completa, **character**, pode ser usada no lugar de **char**.
- **varchar**(n): uma string de caracteres de tamanho variável com tamanho n máximo especificado pelo usuário. A forma completa, **character varying**, é equivalente.
- **int**: um inteiro (um subconjunto finito de inteiros que é dependente da máquina). A forma completa, **integer**, é equivalente.
- **smallint**: um inteiro pequeno (um subconjunto dependente da máquina do tipo de domínio inteiro).
- **numeric**(p,d): um número de ponto fixo com precisão especificada pelo usuário. O número consiste em p dígitos (mais um sinal), e d dos p dígitos estão à direita da vírgula decimal. Assim, **numeric**(3,1) permite que o valor 44,5 seja armazenado exatamente, mas nem 444,5 nem 0,32 podem ser armazenados exatamente em um campo desse tipo.
- **real**, **double precision**: números de ponto flutuante e ponto flutuante de precisão dupla com precisão dependente da máquina.
- **float**(n): um número de ponto flutuante, com precisão de pelo menos n dígitos.

Valores de domínio adicionais são discutidos na seção "Tipos de dados e esquemas da SQL" do Capítulo 4.

agência(<u>*nome_agência*</u>, *cidade_agência*, *ativo*)
cliente (<u>*nome_cliente*</u>, *rua_cliente*, *cidade_cliente*)
empréstimo (<u>*número_empréstimo*</u>, *nome_agência*, *quantia*)
tomador (<u>*nome_cliente*</u>, <u>*número_empréstimo*</u>)
conta (<u>*número_conta*</u>, *nome_agência*, *saldo*)
depositante (<u>*nome_cliente*</u>, <u>*número_conta*</u>)

Figura 3.1 Esquema da instituição bancária.

```
create table cliente
   (nome_cliente  char(20),
    rua_cliente   char(30),
    cidade_cliente char(30),
    primary key (nome_cliente))

create table agência
   (nome_agência  char(15),
    cidade_agência char(30),
    ativo         numeric(16,2),
    primary key (nome_agência))

create table conta
   (número_conta  char(10),
    nome_agência  char(15),
    saldo         muneric(12,2),
    primary key (número_conta))

create table depositante
   (nome_cliente  char(20),
    número_conta  char(10),
    primary key (nome_cliente, número_conta))
```

Figura 3.2 Definição de dados SQL para parte do banco de dados de banco.

Definição básica de esquema na SQL

Definimos uma relação SQL usando o comando **create table**:

create table $r(A_1D_1, A_2D_2, ..., A_nD_n,$
<restrição-de-integridade$_1$>,
...,
<restrição-de-integridade$_k$>)

onde r é o nome da relação, cada A_i é o nome de um atributo no esquema da relação r e D_i é o tipo de domínio dos valores no domínio do atributo A_i. Existem diversas restrições de integridade permitidas. Nesta seção, discutimos apenas a chave primária que tem a forma:

- primary key $(A_{j1}, A_{j2}, ..., A_{jm})$: a especificação de primary key diz que os atributos $A_{j1}, A_{j2}, ..., A_{jm}$ formam a chave primária para a relação. É necessário que os atributos de chave primária sejam *não nulos* e *únicos*; isto é, nenhuma tupla pode ter um valor nulo para um atributo de chave primária, e nenhum par de tuplas na relação pode ser igual em todos os atributos de chave primária.[1] Embora a especificação de chave primária seja opcional, geralmente é uma boa idéia especificar uma chave primária para cada relação.

Outras restrições de integridade que o comando **create table** pode incluir são discutidas mais tarde, na seção "Restrições de integridade" do Capítulo 4.

A Figura 3.2 apresenta uma definição DDL SQL parcial de nosso banco de dados de banco. Note que, como nos capítulos anteriores, não tentamos modelar precisamente a realidade no exemplo de banco de dados de banco. Como, no mundo real, várias pessoas podem ter o mesmo nome, *nome_cliente* não seria uma chave primária *cliente*; um *id_cliente* mais provavelmente seria usado como uma chave primária. Usamos *nome_cliente* como uma chave primária para manter o esquema de banco de dados simples e curto.

Se uma tupla recém-inserida ou modificada em uma relação tiver valores nulos para qualquer atributo de chave primária, ou se a tupla tiver o mesmo valor nos atributos de chave primária que outra tupla na relação, a SQL sinaliza um erro e impede a atualização.

Uma relação recém-criada está inicialmente vazia. Podemos usar o comando **insert** para carregar dados na relação. Por exemplo, se quisermos inserir o fato de que existe uma conta A-9732 na agência Perryridge e que essa conta possui o saldo de $1.200, escrevemos

[1] Na SQL-89, os atributos de chave primária não eram declarados implicitamente como sendo **not null**; uma declaração **not null** explícita era necessária.

```
insert into conta
        values ('A-9732', 'Perryridge', 1200)
```

Os valores são especificados na ordem em que os atributos correspondentes são listados no esquema de relação. O comando insert possui vários recursos úteis e é discutido em mais detalhes posteriormente, na seção "Inserção".

Podemos usar o comando **delete** para excluir tuplas de uma relação. O comando

```
delete from conta
```

excluiria todas as tuplas da relação *conta*. Outras formas do comando delete permitem especificar as tuplas a serem excluídas; o comando delete é abordado em mais detalhes na seção "Exclusão".

Para remover uma relação de um banco de dados SQL, usamos o comando **drop table**. O comando **drop table** exclui do banco de dados todas as informações sobre a relação removida. O comando

```
drop table r
```

é uma ação mais drástica que

```
drop from r
```

O último conserva a relação *r*, mas exclui todas as tuplas em *r*. O primeiro exclui não apenas todas as tuplas de *r*, mas também o esquema para *r*. Depois que *r* é excluído, nenhuma tupla pode ser inserida em *r* a menos que ela seja recriada com o comando **create table**.

O comando **alter table** é usado para acrescentar atributos a uma relação existente. Sua forma é

```
alter table r add A D
```

onde *r* é o nome de uma relação existente, *A* é o nome do atributo a ser acrescentado e *D* é o domínio do atributo acrescentado. Podemos excluir atributos de uma relação pelo comando

```
alter table r drop A
```

onde *r* é o nome de uma relação existente e *A* é o nome de um atributo da relação. Muitos sistemas de banco de dados não aceitam a remoção de atributos, embora permitam que uma tabela inteira seja excluída.

Estrutura básica das consultas SQL

Um banco de dados relacional consiste em uma coleção de relações, cada uma recebendo um nome único. Cada relação possui uma estrutura semelhante à apresentada no Capítulo 2. A SQL permite o uso de valores nulos para indicar que o valor é desconhecido ou inexistente. Ela permite que um usuário especifique os atributos aos quais não podem ser atribuídos valores nulos, como observamos na seção "Definição de dados".

A estrutura básica de uma expressão SQL consiste em três cláusulas: **select**, **from** e **where**.

- A cláusula **select** corresponde à operação projeção da álgebra relacional. Ela é usada para listar os atributos desejados no resultado de uma consulta.
- A cláusula **from** corresponde à operação produto cartesiano da álgebra relacional. Ela lista as relações a serem lidas na avaliação da expressão.
- A cláusula **where** corresponde ao predicado de seleção da álgebra relacional. Ela consiste em um predicado envolvendo atributos das relações que aparecem na cláusula **from**.

Que o termo *select* na SQL possuir um significado diferente do que na álgebra relacional é um fato histórico infeliz. Enfatizamos as diferentes representações aqui para minimizar possíveis confusões.

Uma consulta SQL típica tem a forma

$$\begin{aligned}&\textbf{select } A_1, A_2, \ldots, A_n\\&\textbf{from } r_1, r_2, \ldots, r_m\\&\textbf{where } P\end{aligned}$$

Cada A_i representa um atributo, e cada r_i, uma relação. P é um predicado. A consulta é equivalente à expressão de álgebra relacional

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

Se a cláusula **where** for omitida, o predicado P é **true**. Entretanto, ao contrário de uma expressão de álgebra relacional, o resultado da consulta SQL pode conter múltiplas cópias de algumas tuplas; retornaremos a esse assunto na seção "Duplicatas".

A SQL forma o produto cartesiano das relações nomeadas na cláusula **from**, realiza uma seleção de álgebra relacional usando o predicado da cláusula **where** e, depois, projeta o resultado nos atributos da cláusula **select**. Na prática, a SQL pode converter a expressão em uma forma equivalente que pode ser processada de maneira mais eficiente. Mas deixaremos as preocupações sobre eficiência para os Capítulos 13 e 14.

A cláusula select

O resultado de uma consulta SQL é, evidentemente, uma relação. Vamos considerar uma consulta simples usando

nosso exemplo de banco, "Encontre os nomes de todas as agências na relação *empréstimo*".

select *nome_agência*
from *empréstimo*

O resultado é uma relação consistindo em um único atributo com o cabeçalho *nome_agência*.

As linguagens de consulta formais são baseadas na noção matemática de uma relação sendo um conjunto. Portanto, tuplas duplicadas nunca aparecem em relações. Na prática, eliminação duplicada gasta tempo. Portanto, a SQL (como a maioria das outras linguagens comerciais) permite duplicatas em relações, bem como nos resultados de expressões SQL. Assim, a consulta anterior listará cada *nome_agência* para cada tupla em que ele aparece na relação *empréstimo*.

Nos casos em que queremos forçar a eliminação de duplicatas, inserimos a palavra-chave **distinct** após **select**. Podemos reescrever a consulta anterior como

select distinct *nome_agência*
from *empréstimo*

se quisermos que as duplicatas sejam removidas.

A SQL permite usar a palavra-chave **all** para especificar explicitamente que as duplicatas não devem ser removidas:

select all *nome_agência*
from *empréstimo*

Como a retenção de duplicata é o padrão, não usaremos **all** em nossos exemplos. Para garantir a eliminação das duplicatas nos resultados de nossas consultas de exemplo, usaremos **distinct** sempre que necessário. Na maioria das consultas em que **distinct** não é usado, o número exato de cópias duplicadas de cada tupla presente na consulta não é importante. Entretanto, o número é importante em certas aplicações; retornaremos a essa questão na seção "Duplicatas".

O símbolo de asterisco "*" pode ser usado para indicar "todos os atributos". Portanto, o uso de *empréstimo*.* na cláusula **select** anterior indicaria que todos os atributos de *empréstimo* devem ser selecionados. Uma cláusula **select** da forma **select*** indica que todos os atributos de todas as relações aparecendo na cláusula **from** são selecionados.

A cláusula **select** também pode conter expressões aritméticas envolvendo os operadores +, –, * e / operando em constantes ou atributos de tuplas. Por exemplo, a consulta

select all *número_empréstimo, nome_agência, quantia* * 100
from *empréstimo*

retornará uma relação que é a mesma relação *empréstimo*, exceto que o atributo *quantia* é multiplicado por 100.

A SQL também fornece tipos de dados especiais, como várias formas do tipo *date*, e permite que várias funções aritméticas operem nesses tipos.

A cláusula where

Vamos ilustrar o uso da cláusula **where** na SQL. Considere a consulta "Encontre todos os números de empréstimo para empréstimos feitos na agência Perryridge com quantias superiores a $1200". Essa consulta pode ser escrita em SQL como:

select *número_empr*
from *empréstimo*
where *nome_agência* = 'Perryridge' **and** *quantia* > 1200

A SQL usa os conectivos **and**, **or** e **not** – em vez dos símbolos matemáticos $\wedge$, $\vee$ e $\neg$ – na cláusula **where**. Os operandos dos conectivos lógicos podem ser expressões envolvendo os operadores de comparação <, <=, >, >=, = e <>. A SQL permite usar os operadores de comparação para comparar strings e expressões aritméticas além de tipos especiais, como tipos de data.

A SQL inclui um operador de comparação **between** para simplificar cláusulas **where** que especificam que um valor seja menor ou igual a algum valor e maior ou igual a algum outro valor. Se quisermos encontrar o número dos empréstimos com quantias entre $90.000 e $100.000, podemos usar o operador de comparação **between** para escrever

select *número_empr*
from *empréstimo*
where *quantia* **between** 90000 **and** 100000

em vez de

select *número_empr*
from *empréstimo*
where *quantia* <= 100000 **and** amount >= 90000

Da mesma forma, podemos usar o operador de comparação **not between**.

A cláusula from

Finalmente, vamos discutir o uso da cláusula **from**. A cláusula **from** isolada define um produto cartesiano das relações na cláusula. Já que a junção natural é definida em termos de um produto cartesiano, uma seleção e uma projeção, é relativamente simples escrever uma expressão para a junção natural.

Escrevemos a expressão de álgebra relacional

$\Pi_{nome\_cliente,\ número\_empréstimo,\ conta}$ (*tomador* ⋈ *empréstimo*)

para a consulta "Para todos os clientes que têm um empréstimo do banco, encontre seus nomes, números de empréstimo e quantia do empréstimo". Na SQL, essa consulta pode ser escrita como

select *nome_cliente, tomador.número_empréstimo, quantia*
from *tomador, empréstimo*
where *tomador.número_empréstimo = empréstimo.número_empréstimo*

Observe que a SQL usa a notação nome-relação.nome-atributo, como a álgebra relacional, para evitar ambigüidade nos casos em que um atributo aparece no esquema de mais de uma relação. Poderíamos ter escrito *tomador.nome_cliente* em vez de *nome_cliente* na cláusula **select**. Entretanto, como o atributo *nome_cliente* aparece apenas em uma das relações nomeadas na cláusula **from**, não há ambigüidade quando escrevemos *nome_cliente*.

Podemos estender a consulta anterior e considerar um caso mais complicado em que também exigimos que o empréstimo seja da agência Perryridge: "Encontre os nomes de cliente, os números de empréstimo e as quantias de empréstimo para todos os empréstimos da agência Perryridge". Para escrever essa consulta, é preciso informar as duas restrições na cláusula **where**, conectadas pelo conectivo lógico **and**:

select *nome_cliente, tomador.número_empréstimo, quantia*
from *tomador, empréstimo*
where *tomador.número_empréstimo = empréstimo.número_empréstimo* **and**
nome_agência = 'Perryridge'

A SQL inclui extensões para realizar junções naturais e junções externas na cláusula **from**. Discutiremos essas extensões na seção "Relações juntadas**".

A operação de renomeação

A SQL fornece um mecanismo para renomear relações e também atributos. Ela usa a cláusula **as**, tomando a forma:

nome-antigo **as** *nome-novo*

A cláusula **as** pode aparecer tanto na cláusula **select** quanto na cláusula **from**.

Considere novamente a consulta que usamos anteriormente:

select *nome_cliente, tomador.número_empréstimo, quantia*
from *tomador, empréstimo*
where *tomador.número_empréstimo = empréstimo.número_empréstimo*

O resultado dessa consulta é uma relação com os seguintes atributos:

nome_cliente, número_empréstimo, quantia

Os nomes dos atributos no resultado são derivados dos nomes dos atributos nas relações na cláusula **from**.

Entretanto, nem sempre podemos derivar nomes dessa forma, por várias razões. Primeiro, duas relações na cláusula **from** podem ter atributos com o mesmo nome, caso em que um nome de atributo é duplicado no resultado. Segundo, se usarmos uma expressão aritmética na cláusula **select**, o atributo resultante não terá um nome. Terceiro, mesmo que um nome de atributo possa ser derivado das relações de base, como no exemplo anterior, podemos querer mudar o nome de atributo no resultado. Por isso, a SQL fornece uma maneira de renomear os atributos de uma relação resultante.

Por exemplo, se quisermos que o nome de atributo *número_empréstimo* seja substituído pelo nome *id_empréstimo*, podemos reescrever a consulta anterior como

select *nome_cliente, tomador.número_empréstimo* **as** *id_empréstimo, quantia*
from *tomador, empréstimo*
where *tomador.número_empréstimo = empréstimo.número_empréstimo*

Variáveis de tupla

A cláusula **as** é particularmente útil para definir a noção de variáveis de tupla. Uma variável de tupla na SQL precisa estar associada a uma determinada relação. As variáveis de tupla são definidas na cláusula **from** por meio da cláusula **as**. Para ilustrar, escrevemos a consulta "Para todos os clientes que têm um empréstimo do banco, encontre seus nomes, números de empréstimo e quantia do empréstimo" como

select *nome_cliente, T.número_empréstimo, S.quantia*
from *tomador* **as** *T, empréstimo* **as** *S*
where *T.número_empréstimo = S.número_empréstimo*

Observe que definimos uma variável de tupla na cláusula **from** colocando-a após o nome da relação à qual ela está associada, com a palavra-chave **as** entre as duas (a palavra-chave **as** é opcional). Quando escrevemos expressões da forma *nome-relação.nome-atributo*, o nome da relação é, na verdade, uma variável de tupla definida implicitamente.

As variáveis de tupla são mais úteis para comparar duas tuplas na mesma relação. Lembre-se de que, nesses casos, poderíamos usar a operação de renomeação na álgebra relacional. Suponha que queremos realizar a consulta "Encontre os nomes de todas as agências que têm ativos maiores do que pelo menos uma agência localizada em Brooklyn". Podemos escrever a expressão SQL

select distinct *T.nome_agência*
from *agência* **as** *T*, *agência* **as** *S*
where *T.ativo* > *S.ativo* **and** *S.cidade_agência* = 'Brooklyn'

Observe que não poderíamos usar a notação *agência.ativo*, já que não ficaria claro que referência a *agência* é desejada.

A SQL nos permite usar a notação $(v_1, v_2, ..., v_n)$ para indicar uma tupla de aridade n contendo valores $v_1, v_2, ..., v_n$. Os operadores de comparação podem ser usados em tuplas, e a ordenação é definida lexicograficamente. Por exemplo, $(a_1, a_2) <= (b_1, b_2)$ é verdadeiro se $a_1 < b_1$ ou $(a_1 = b_1) \wedge (a_2 <= b_2)$; da mesma forma, as duas tuplas são iguais se todos os seus atributos forem iguais.

Operações de string

A SQL especifica strings inserindo-as entre sinais de apóstrofo, por exemplo, 'Perryridge', como vimos anteriormente. Um caractere de apóstrofo que seja parte de uma string pode ser especificado usando dois caracteres de apóstrofo; por exemplo, a string "Caixa d'água" pode ser especificada por "Caixa d''água".

A operação mais usada em strings é a correspondência de padrões usando o operador **like**. Descrevemos os padrões usando dois caracteres especiais:

- Porcentagem (%): o caractere % corresponde a qualquer substring.
- Sublinhado (_): o caractere _ corresponde a qualquer caractere.

Os padrões fazem distinção entre letras maiúsculas e minúsculas; ou seja, caracteres maiúsculos não são o mesmo que caracteres minúsculos ou vice-versa. Para ilustrar a correspondência de padrões, considere os seguintes exemplos:

- 'Perry%' localiza qualquer string começando com "Perry".
- '%idge%' localiza qualquer string contendo "idge" como uma substring, por exemplo, 'Perryridge', 'Rock Ridge', 'Mianus Bridge' e 'Ridgeway'.
- '_ _ _' localiza qualquer string com exatamente três caracteres.
- '_ _ _%' localiza qualquer string com pelo menos três caracteres.

A SQL expressa padrões usando o operador de comparação **like**. Considere a consulta "Encontre os nomes de todos os clientes cujos endereços de rua incluem a substring 'Main'." Essa consulta pode ser escrita como

select *nome_cliente*
from *cliente*
where *rua_cliente* **like** '%Main%'

Para que os padrões incluam os caracteres de padrão especiais (ou seja, % e _), a SQL permite a especificação de um caractere de escape. O caractere de escape é usado imediatamente antes de um caractere de padrão especial para indicar que este deve ser tratado como um caractere normal. Definimos o caractere de escape para uma comparação **like** usando a palavra-chave **escape**. Para ilustrar, considere os seguintes padrões, que usam uma barra invertida (\) como caractere de escape:

- **like** 'ab\%cd%' **escape** '\' localiza todas as strings iniciando com "ab%cd".
- **like** 'ab\\cd%' **escape** '\' localiza todas as strings iniciando com "ab\cd".

A SQL permite pesquisar disparidades em vez de correspondências usando o operador de comparação **not like**.

A SQL também permite uma variedade de funções em strings de caractere, como concatenação (usando "||"), extração de substrings, localização do tamanho de strings, conversão de strings em maiúsculas (usando **upper**( )) e minúsculas (usando **lower**( )) e assim por diante. A SQL:1999 também oferece uma operação **similar to**, que fornece uma correspondência de padrões mais poderosa do que a operação **like**; a sintaxe para especificar padrões é semelhante à usada em expressões regulares do UNIX.

Existem variações no conjunto exato de funções de string aceitas por diferentes sistemas de banco de dados. Alguns sistemas de banco de dados não fazem distinção entre maiúsculas e minúsculas quando localizam strings. Assim, 'ABC' **like** 'abc' retornaria true, bem como 'ABC' = 'abc' nesses sistemas. Outros fornecem extensões para especificar que uma correspondência de string ignoraria se os caracteres são maiúsculos ou minúsculos. Veja o manual do seu sistema de banco de dados para obter mais detalhes sobre que funções de string exatamente ele aceita.

Ordenação da exibição de tuplas

A SQL oferece ao usuário algum controle sobre a ordem em que as tuplas são exibidas em uma relação. A cláusula **order by** faz com que as tuplas no resultado de uma consulta apareçam na ordem classificada. Para listar em ordem alfabética todos os clientes que têm um empréstimo na agência Perryridge, escrevemos

select *nome_cliente*
from *tomador, empréstimo*
where *tomador.número_empréstimo = empréstimo.número_empréstimo* **and** *nome_agência* = 'Perryridge'
order by *nome_cliente*

Como padrão, a cláusula **order by** lista itens na ordem crescente. Para especificar a ordem de classificação, podemos especificar **desc** para ordem decrescente ou **asc** para ordem crescente. Além disso, a ordenação pode ser realizada em vários atributos. Suponha que desejamos listar a relação *empréstimo* inteira na ordem decrescente de *quantia*. Se vários empréstimos possuem a mesma quantia, o ordenamos na ordem crescente por número de empréstimo. Expressamos essa consulta na SQL assim:

select *
from *empréstimo*
order by *quantia* **desc**, *número_empréstimo* asc

Para satisfazer uma requisição **order by**, a SQL precisa realizar uma classificação. Como classificar um grande número de tuplas pode ser custoso, isso deve ser feito apenas quando necessário.

Duplicatas

O uso de relações com duplicatas oferece vantagens em várias situações. Por isso, a SQL define formalmente não apenas quais tuplas estão no resultado de uma consulta, mas também como muitas cópias de cada uma dessas tuplas aparecem no resultado. Podemos definir a semântica de duplicata de uma consulta SQL usando versões *multiconjunto* dos operadores relações. Aqui, definimos as versões multiconjunto de várias operações de álgebra relacional. Dadas as relações multiconjunto r_1 e r_2,

1. Se existem c_1 cópias da tupla t_1 em r_1, e t_1 satisfaz a seleção σ_θ, então, existem c_1 cópias de t_1 em $\sigma_\theta(r_1)$.
2. Para cada cópia da tupla t_1 em r_1, existe uma cópia da tupla $\Pi_A(t_1)$ em $\Pi_A(r_1)$, onde $\Pi_A(t_1)$ indica a projeção da tupla única t_1.
3. Se existem c_1 cópias da tupla t_1 em r_1 e c_2 cópias da tupla t_2 em r_2, então, existem $c_1 * c_2$ cópias da tupla $t_1.t_2$ em $r_1 \times r_2$.

Por exemplo, suponha que as relações r_1 com o esquema (A,B) e r_2 com o esquema (C) sejam os seguintes multiconjuntos:

$$r_1 = \{(1,a), (2,a)\} \quad r_2 = \{(2), (3), (3)\}$$

Então, $\Pi_B(r_1)$ seria $\{(a), (a)\}$, enquanto $\Pi_B(r_1) \times r_2$ seria

$$\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$$

Agora, podemos definir quantas cópias de cada tupla ocorrem no resultado de uma consulta SQL. Uma consulta SQL da forma

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P

é equivalente à expressão da álgebra relacional

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

usando as versões de multiconjunto dos operadores relacionais σ, Π e X.

Operações de conjunto

As operações SQL **union**, **intersect** e **except** operam em relações e correspondem às operações da álgebra relacional $\cup$, $\cap$ e $-$. Assim como as operações união, interseção e diferença de conjuntos na álgebra relacional, as relações que participam das operações precisam ser *compatíveis*; isto é, precisam ter o mesmo conjunto de atributos.

Vamos demonstrar como várias consultas de exemplo vistas no Capítulo 2 podem ser escritas na SQL. Agora construiremos consultas envolvendo as operações **union**, **intersect** e **except** de dois conjuntos: o conjunto de todos os clientes que têm uma conta no banco, que pode ser derivado por

select *nome_cliente*
from *depositante*

e o conjunto dos clientes que têm um empréstimo no banco, que pode ser derivado por

select *nome_cliente*
from *tomador*

Em nossa discussão que se segue, iremos nos referir às relações obtidas como resultado das consultas anteriores como d e b, respectivamente.

A operação union

Para encontrar todos os clientes do banco que possuem um empréstimo, uma conta ou as duas coisas no banco, escrevemos

```
(select nome_cliente
from depositante)
union
(select nome_cliente
from tomador)
```

A operação **union** elimina automaticamente as duplicatas, ao contrário da cláusula **select**. Portanto, na consulta anterior, se um cliente – digamos, Jones – possuir várias contas ou empréstimos (ou ambos) no banco, então, Jones aparecerá apenas uma vez no resultado.

Se quisermos manter todas as duplicatas, precisamos escrever **union all** no lugar de **union**:

```
(select nome_cliente
from depositante)
union all
(select nome_cliente
from tomador)
```

O número de tuplas duplicadas no resultado é igual ao número total de duplicatas que aparecem em *d* e *b*. Portanto, se Jones tiver três contas e dois empréstimos no banco, então, haverá cinco tuplas com o nome Jones no resultado.

A operação intersect

Para encontrar todos os clientes que possuem um empréstimo e uma conta no banco, escrevemos

```
(select distinct nome_cliente
from depositante)
intersect
(select distinct nome_cliente
from tomador)
```

A operação **insersect** elimina automaticamente as duplicatas. Assim, na consulta anterior, se um cliente – digamos, Jones – tiver várias contas e empréstimos no banco, então, Jones aparecerá apenas uma vez no resultado.

Se quisermos manter todas as duplicatas, precisamos escrever **intersect all** no lugar de **intersect**:

```
(select nome_cliente
from depositante)
intersect all
(select nome_cliente
from tomador)
```

O número de tuplas duplicadas que aparecem no resultado é igual ao número mínimo de duplicatas em *d* e *b*. Assim, se Jones tiver três contas e dois empréstimos no banco, então, haverá duas tuplas com o nome Jones no resultado.

A operação except

Para encontrar todos os clientes que possuem uma conta mas nenhum empréstimo no banco, escrevemos

```
(select distinct nome_cliente
from depositante)
except
(select nome_cliente
from tomador)
```

A operação **except** elimina automaticamente as duplicatas. Assim, na consulta anterior, uma tupla com o nome de cliente Jones aparecerá (exatamente uma vez) no resultado apenas se Jones tiver uma conta no banco, mas nenhum empréstimo no banco.

Se quisermos manter todas as duplicatas, precisamos escrever **except all** no lugar de **except**:

```
(select nome_cliente
from depositante)
except all
(select nome_cliente
from tomador)
```

O número de cópias duplicadas de uma tupla no resultado é igual ao número de cópias duplicadas da tupla em *depositante* menos o número de cópias duplicadas da tupla em *tomador*, desde que a diferença seja positiva. Assim, se Jones tiver três contas e um empréstimo no banco, então, haverá duas tuplas com o nome Jones no resultado. Se, em vez disso, esse cliente tiver duas contas e três empréstimos no banco, então, não haverá tupla alguma com o nome Jones no resultado.

Funções agregadas

Funções agregadas são aquelas que tomam uma coleção (um conjunto ou multiconjunto) de valores como entrada e retornam um único valor. A SQL oferece cinco funções agregadas embutidas:

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

A entrada para **sum** e **avg** precisa ser uma coleção de números, mas os outros operadores podem operar também em coleções de tipos de dados não numéricos, como strings.

Como uma ilustração, considere a consulta "Encontre o saldo médio das contas na agência Perryridge", que escrevemos desta forma:

```
select avg (saldo)
from conta
where nome_agência = 'Perryridge'
```

O resultado dessa consulta é uma relação com um único atributo, contendo uma única tupla com um valor numérico correspondente ao saldo médio na agência Perryridge. Opcionalmente, podemos dar um nome ao atributo da relação resultante usando a cláusula **as**.

Em algumas situações desejaríamos aplicar a função agregada não apenas a um único conjunto de tuplas, mas também a um grupo de conjuntos de tuplas; na SQL, isso é feito usando a cláusula **group by**. O atributo ou atributos fornecidos na cláusula **group by** são usados para formar grupos. As tuplas com o mesmo valor em todos os atributos na cláusula **group by** são colocadas em um grupo.

Como uma ilustração, considere a consulta "Encontre o saldo médio em cada agência", que escrevemos assim:

```
select nome_agência, avg (saldo)
from conta
group by nome_agência
```

É importante manter as duplicatas no cálculo de uma média. Suponha que os saldos das contas na (pequena) agência Brighton sejam $1.000, $3.000, $2.000 e $1.000. O saldo médio, portanto, é $7.000/4 = $1.750. Se as duplicatas fossem eliminadas, obteríamos a resposta errada ($6.000/3 = $2.000).

Existem casos em que precisamos eliminar duplicatas antes de calcular uma função agregada. Se realmente quisermos eliminar duplicatas, usamos a palavra-chave **distinct** na expressão agregada. Um exemplo surge na consulta "Encontre o número de depositantes de cada agência". Nesse caso, um depositante conta apenas uma vez, independentemente do número de contas que ele possa ter. Escrevemos essa consulta da seguinte maneira:

```
select nome_agência, count (distinct nome_cliente)
from depositante, conta
where depositante.número_conta = conta.número_conta
group by nome_agência
```

Às vezes, é útil informar uma condição que se aplica a grupos em vez de tuplas. Por exemplo, podemos estar interessados apenas nas agências em que o saldo médio é maior que $1.200. Essa condição não se aplica a uma única tupla, mas sim em cada grupo construído pela cláusula **group by**. Para expressar essa consulta, usamos a cláusula **having** da SQL. A SQL aplica predicados na cláusula **having** após os grupos terem sido formados, permitindo, assim, o uso de funções agregadas. Expressamos essa consulta na SQL desta forma:

```
select nome_agência, avg (saldo)
from conta
group by nome_agência
having avg (saldo) > 1200
```

Ocasionalmente, desejamos tratar a relação inteira como um único grupo. Nesses casos, não usamos uma cláusula **group by**. Considere a consulta "Encontre o saldo médio para todas as contas", que escrevemos assim:

```
select avg (saldo)
from conta
```

Usamos a função agregada **count** freqüentemente para contar o número de tuplas em uma relação. A notação para essa função na SQL é **count** (*). Portanto, para encontrar o número de tuplas na relação *cliente*, escrevemos

```
select count (*)
from cliente
```

A SQL não permite o uso de **distinct** com **count** (*). É válido usar **distinct** com **max** e **min**, mesmo se o resultado não mudar. Podemos usar a palavra-chave **all** no lugar de **distinct** para especificar retenção de duplicata, mas, como **all** é o padrão, não há necessidade de fazer isso.

Se uma cláusula **where** e uma cláusula **having** aparecerem na mesma consulta, a SQL aplica o predicado na cláusula **where** primeiro. As tuplas que satisfazem o predicado **where** são, então, colocadas em grupos pela cláusula **group by**. A SQL, então, aplica a cláusula **having**, se presente, a cada grupo; ela remove os grupos que não satisfazem o predicado da cláusula **having**. A cláusula **select** usa os grupos restantes para gerar tuplas do resultado da consulta.

Para ilustrar o uso da cláusula **having** e da cláusula **where** na mesma consulta, consideramos a consulta "Encontre o saldo médio para cada cliente que mora em Harrison e tem pelo menos três contas".

```
Select depositante.nome_cliente, avg (saldo)
from depositante, conta, cliente
where depositante.número_conta = conta.número_conta and
depositante.nome_cliente = cliente.nome_cliente and
    cidade_cliente = 'Harrison'
group by depositante.nome_cliente
having count (distinct depositante.número_conta) >= 3
```

Valores nulos

A SQL permite o uso de valores *nulos* para indicar a ausência de informações sobre o valor de um atributo.

Podemos usar a palavra-chave especial **null** em um predicado para testar a presença de um valor nulo. Portanto, para encontrar todos os números de empréstimo na relação *empréstimo* com valores nulos para *conta*, escrevemos

```
select número_empréstimo
from empréstimo
where quantia is null
```

O predicado **is not null** testa a ausência de um valor nulo.

O uso de um valor *nulo* em operações aritméticas e de comparação causa várias complicações. Na seção "Valores nulos" do Capítulo 2 vimos como os valores nulos são manipulados na álgebra relacional. Agora, descrevemos como a SQL manipula valores nulos.

O resultado de uma expressão aritmética (envolvendo, por exemplo, +, –, * ou /) é nulo se qualquer um dos valores de entrada for nulo. A SQL trata como **unknown** (desconhecido) o resultado de qualquer comparação envolvendo um valor *nulo* (que não **is null** e **is not null**).

Como o predicado em uma cláusula **where** pode envolver operações booleanas como **and**, **or** e **not** nos resultados de comparações, as definições das operações booleanas são estendidas para lidar com o valor **unknown**, como descrito na seção "Valores nulos" do Capítulo 2.

- **and**: o resultado de *true* **and** *unknown* é *unknown*; o de *false* **and** *unknown* é *false*; enquanto o de *unknown* **and** *unknown* é *unknown*.
- **or**: o resultado de *true* **or** *unknown* é *true*; o de *false* **or** *unknown* é *unknown*; enquanto o de *unknown* **or** *unknown* é *unknown*.
- **not**: o resultado de **not** *unknown* é *unknown*.

A SQL define o resultado de uma instrução SQL da forma

select ... **from** $R_1, \ldots, R_n$ **where** P

para conter (projeções de) tuplas em $R_1 \times \ldots \times R_n$ para qual predicado P é avaliado para **true**. Se o predicado é avaliado para **false** ou **unknown** para uma tupla em $R_1 \times \ldots \times R_n$, (a projeção de) a tupla não é acrescentada ao resultado.

A SQL também permite testar se o resultado de uma comparação é desconhecido, em vez de verdadeiro ou falso, usando as cláusulas **is unknown** e **is not unknown**.

Os valores nulos, quando existem, também complicam o processamento dos operadores agregados. Por exemplo, considere que algumas tuplas na relação *empréstimo* tenham um valor nulo para *quantia*. Considere a seguinte consulta para somar todas as quantias de empréstimo:

```
select sum (quantia)
from empréstimo
```

Os valores a serem somados nessa consulta incluem valores nulos, já que algumas tuplas possuem um valor nulo para *quantia*. Em vez de dizer que a soma total é, ela mesma, *nula*, o padrão SQL diz que o operador **sum** deve ignorar valores *nulos* em sua entrada.

Em geral, as funções agregadas tratam os nulos de acordo com a seguinte regra: todas as funções agregadas exceto **count**(*) ignoram valores nulos em sua coleção de entrada. Como resultado dos valores nulos sendo ignorados, a coleção de valores pode ser vazia. O **count** de uma coleção vazia é definida para ser 0 e todas as outras operações agregadas retornam um valor de nulo quando aplicadas em uma coleção vazia. O efeito de valores nulos em algumas das construções SQL mais complexas pode ser sutil.

Um tipo de dados **boolean**, que pode assumir valores **true**, **false** e **unknown**, foi introduzido na SQL:1999. As funções agregadas **some** e **every**, que significam exatamente o que você já imaginava, podem ser aplicadas em uma coleção de valores booleanos.

Subconsultas aninhadas

A SQL fornece um mecanismo para aninhar subconsultas. Uma subconsulta é uma expressão **select-from-where** que é aninhada dentro de outra consulta. Um uso comum de subconsultas é realizar testes para participação de conjuntos, fazer comparações de conjuntos e determinar cardinalidade de conjuntos. Estudaremos esses usos nas seções que se seguem.

Participação de conjuntos

A SQL permite testar tuplas quanto à participação em uma relação. O conectivo **in** testa a presença em um conjunto, em que o conjunto é uma coleção de valores produzidos por uma cláusula select. O conectivo **not in** testa a ausência em um conjunto.

Como ilustração, considere novamente a consulta "Encontre todos os clientes que possuem um empréstimo e uma conta no banco". Anteriormente, escrevemos essa consulta fazendo a interseção de dois conjuntos: o conjunto dos depositantes do banco e o conjunto dos tomadores de empréstimo do banco. Podemos usar um método alternativo para encontrar todos os correntistas que são membros do conjunto de tomadores de empréstimo do banco. Claramente, essa formulação gera os mesmos resultados

que a formulação anterior gerou, mas ela nos leva a escrever nossa consulta usando o conectivo **in** da SQL. Começamos encontrando todos os correntistas do banco, escrevendo a subconsulta

```
(select nome_cliente
from depositante)
```

Depois, precisamos encontrar os clientes que são tomadores de empréstimo do banco e que aparecem na lista de correntistas obtida na subconsulta. Fazemos isso aninhando a subconsulta em um **select** externo. A consulta resultante é

```
select distinct nome_cliente
from tomador
where nome_cliente in (select nome_cliente
                       from depositante)
```

Esse exemplo mostra que é possível escrever a mesma consulta de várias maneiras na SQL. Essa flexibilidade é benéfica, já que permite que um usuário pense na consulta da maneira que lhe pareça mais natural. Veremos que existe uma quantidade substancial de redundância na SQL.

No exemplo anterior, testamos participação em uma relação de um atributo. Também é possível testar a participação em uma relação arbitrária na SQL. Portanto, podemos escrever a consulta "Encontre todos os clientes que possuem uma conta e um empréstimo na agência Perryridge" ainda de outra maneira:

```
select distinct nome_cliente
from tomador, empréstimo
where tomador.número_cliente = empréstimo.número_
      empréstimo and
          nome_agência = 'Perryridge' and
          (nome_agência, nome_cliente) in
            (select nome_agência, nome_cliente
            from depositante, conta
             where depositante.número_conta = conta.
            número_conta)
```

Usamos a construção **not in** de uma maneira semelhante. Por exemplo, para encontrar todos os clientes que possuem um empréstimo no banco, mas que não têm uma conta no banco, podemos escrever

```
select distinct nome_cliente
from tomador
where nome_cliente) not in (select nome_cliente
                            from depositante)
```

Os operadores **in** e **not in** também podem ser usados em conjuntos enumerados. A consulta a seguir seleciona os clientes que têm um empréstimo no banco e cujos nomes não são nem Smith nem Jones.

```
select distinct nome_cliente
from tomador
where nome_cliente) not in ('Smith', 'Jones')
```

Comparação de conjuntos

Como um exemplo da capacidade de uma subconsulta aninhada comparar conjuntos, considere a consulta "Encontre os nomes de todas as agências que possuem ativos maiores do que o ativo de pelo menos uma agência localizada em Brooklyn". Na seção "Variáveis de tupla", escrevemos essa consulta da seguinte maneira:

```
select distinct T.nome_agência
from agência as T, agência as S
where T.ativo > S.ativo and S.cidade_agência = 'Brooklyn'
```

Entretanto, a SQL oferece um estilo alternativo de escrever essa consulta. A frase "maior do que pelo menos uma" é representada na SQL por **> some**. Essa construção permite reescrever a consulta em uma forma que se assemelha de perto a nossa formulação da consulta em português.

```
select nome_agência
from agência
where ativo > some (select ativo
                    from agência
                    where cidade_agência = 'Brooklyn')
```

A subconsulta

```
(select ativo
    from agência
    where cidade_agência = 'Brooklyn')
```

gera o conjunto de todos os valores de ativo para todas as agências em Brooklyn. A comparação **> some** na cláusula **where** do **select** externo é verdadeira se o valor de *ativo* da tupla for maior que pelo menos um membro do conjunto de todos os valores de ativo para agências em Brooklyn.

A SQL também permite as comparações **< some**, **<= some**, **>= some**, **= some** e **< > some**. Como um exercício, verifique que **= some** é idêntico a **in**, enquanto **< > some** *não* é o mesmo que **not in**. A palavra-chave **any** é sinônima de **some** na SQL. As versões anteriores da SQL permitiam apenas **any**. As últimas versões acrescentaram a alternativa **some** para evitar a ambigüidade lingüística da palavra *any* no inglês.

Agora modificamos ligeiramente nossa consulta. Vamos encontrar os nomes de todas as agências que possuem um valor de ativo maior que o ativo de cada agência em Brooklyn. A construção **> all** corresponde à frase "maior do que todos". Usando essa construção, escrevemos a consulta desta maneira:

```
select nome_agência
from agência
where ativo > all (select ativo
                   from agência
                   where cidade_agência = 'Brooklyn')
```

A exemplo do que ocorre com **some**, a SQL permite as comparações **< all**, **<= all**, **>= all**, **= all** e **< > all**. Como um exercício, verifique que **< > all** é idêntico a **not in**.

Como outro exemplo de comparações de conjunto, considere a consulta "Encontre a agência que possui o maior saldo médio". As funções agregadas não podem ser compostas na SQL. Portanto, não podemos usar **max** (**avg** (...)). Em vez disso, podemos seguir esta estratégia: começamos escrevendo uma consulta para encontrar todos os saldos médios e, depois, a aninhamos como uma subconsulta de uma consulta maior que encontra as agências para as quais o saldo médio é maior ou igual a todos os saldos médios:

```
select nome_agência
from conta
group by nome_agência
having avg (saldo) >= all (select avg (saldo)
                           from conta
                           group by nome_agência)
```

Teste de relações vazias

A SQL inclui um recurso para testar se uma subconsulta possui alguma tupla no resultado. A construção **exists** retorna o valor **true** se a subconsulta de argumento não é vazia. Usando a construção **exists**, podemos escrever a consulta "Encontre todos os clientes que possuem uma conta e um empréstimo no banco" ainda de outra maneira:

```
select nome_cliente
from tomador
where exists (select *
              from depositante
              where depositante.nome_cliente = tomador.
                    nome_cliente)
```

Podemos testar a inexistência de tuplas em uma subconsulta usando a construção **not exists**, e usar a construção **not exists** para simular a operação confinamento de conjunto (ou seja, superconjunto): é possível escrever "a relação *A* contém a relação *B*" como "**not exists** (*B* **except** *A*)". (Embora não sendo parte dos padrões SQL-92 e SQL:1999, o operador **contains** estava presente em alguns dos primeiros sistemas relacionais.) Para ilustrar o operador **not exists**, considere novamente a consulta "Encontre todos os clientes que têm uma conta em todas as agências localizadas em Brooklyn". Para cada cliente, precisamos ver se o conjunto de todas as agências em que esse cliente tem conta contém o conjunto de todas as agências em Brooklyn. Usando a construção **except**, podemos escrever a consulta da seguinte forma:

```
select distinct S.nome_cliente
from depositante as S
where not exists ((select nome_agência
                   from agência
                   where cidade_agência = 'Brooklyn')
                   except
                      (select R.nome_agência
                      from depositante as T, conta as R
                      where T.número_conta = R.número
                        _conta and
                        S.nome_cliente = T.nome_cliente))
```

Aqui, a subconsulta

```
(select nome_agência
 from agência
 where cidade_agência = 'Brooklyn')
```

encontra todas as agências em Brooklyn. A subconsulta

```
(select R.nome_agência
 from depositante as T, conta as R
 where T.numero_conta = R.numero_conta and
       S.nome_cliente = T.nome_cliente)
```

encontra todas as agências em que o cliente *S.nome_cliente* tem uma conta. Portanto, a cláusula **select** externa toma cada cliente e testa se o conjunto de todas as agências em que esse cliente tem uma conta contém o conjunto de todas as agências localizadas em Brooklyn.

Em consultas que contêm subconsultas, uma regra de escopo se aplica para variáveis de tupla. Em uma subconsulta, de acordo com a regra, é válido usar apenas variáveis de tupla definidas na própria subconsulta ou em qualquer consulta que contenha a subconsulta. Se uma variável de tupla é definida tanto localmente em uma subconsulta quanto globalmente em uma superconsulta, a definição local se aplica. Essa regra é análoga às regras de escopo comuns usadas para variáveis nas linguagens de programação.

Teste da ausência de tuplas duplicatas

A SQL inclui um recurso para testar se uma subconsulta possui alguma tupla duplicata em seu resultado. A construção **unique** retorna o valor **true** se a subconsulta de argumento não contiver qualquer tupla duplicata. Usando a construção **unique**, podemos escrever a consulta "Encontre todos os clientes que possuem no máximo uma conta na agência Perryridge" desta forma:

```
select T.nome_cliente
from depositante as T
where unique (select R.nome_cliente
              from conta, depositante as R
              where T.nome_cliente = R.nome_cliente and
                    R.número_conta = conta.número_conta and
                    conta.nome_agência = 'Perryridge')
```

Podemos testar a existência de tuplas duplicatas em uma subconsulta usando a construção **not unique**. Para ilustrar essa construção, considere a consulta "Encontre todos os clientes que têm pelo menos duas contas na agência Perryridge", que escrevemos como

```
select distinct T.nome_cliente
from depositante as T
where not unique (select R.nome_cliente
                  from conta, depositante as R
                  where T.nome_cliente = R.nome_cliente
                  and R.número_conta = conta.número_
                  conta and conta.nome_agência = 'Perryridge')
```

Formalmente, o teste **unique** em uma relação é definido para falhar se e somente se a relação contém duas tuplas t_1 e t_2 tal que $t_1 = t_2$. Como o teste $t_1 = t_2$ falha se qualquer um dos campos de t_1 ou t_2 for nulo, é possível que **unique** seja verdadeiro, mesmo se houver várias cópias de uma tupla, desde que pelo menos um dos atributos da tupla seja nulo.

Consultas complexas

As consultas complexas normalmente são difíceis ou impossíveis de escrever como um único bloco SQL ou uma união/interseção/diferença de blocos SQL. (Um bloco SQL consiste em uma única instrução **select-from-where**, possivelmente com cláusulas **group by** e **having**.) Estudamos aqui duas maneiras de compor múltiplos blocos SQL para expressar uma consulta complexa: as relações derivadas e a cláusula **with**.

Relações derivadas

A SQL permite que uma expressão de subconsulta seja usada na cláusula **from**. Se usarmos essa expressão, então precisamos dar um nome à relação resultado, e podemos renomear os atributos. Fazemos isso usando a cláusula **as**. Por exemplo, considere a subconsulta

```
(select nome_agência, avg (saldo)
from conta
group by nome_agência)
as média_agência (nome_agência, saldo_médio)
```

Essa subconsulta gera uma relação consistindo nos nomes de todas as agências e seus saldos médios correspondentes. O resultado da subconsulta é denominado *média_agência*, com os atributos *nome_agência* e *saldo_médio*.

Para ilustrar o uso de uma expressão de subconsulta na cláusula **from**, considere a consulta "Encontre o saldo médio das agências onde o saldo médio é maior que $1.200". Escrevemos essa consulta na seção "Funções agregadas" usando a cláusula **having**. Agora podemos reescrever essa consulta sem usar a cláusula **having**, desta maneira:

```
select nome_agência, saldo_médio
from (select nome_agência, avg (saldo)
      from conta
      group by nome_agência)
      as média_agência (nome_agência, saldo_médio)
where saldo_médio > 1200
```

Repare que não foi preciso usar a cláusula **having**, já que a subconsulta na cláusula **from** calcula o saldo médio e seu resultado é nomeado como *média_agência*; podemos usar os atributos de *média_agência* diretamente na cláusula **where**.

Como outro exemplo, suponha que desejamos encontrar o máximo entre todas as agências do saldo total em cada agência. A cláusula **having** não nos ajuda nessa tarefa, mas podemos escrever essa consulta facilmente usando uma subconsulta na cláusula **from**, desta maneira:

```
select max (saldo_total)
from (select nome_agência, sum (saldo)
      from conta
      group by nome_agência) as total_agência (nome_
      agência, saldo_total)
```

A cláusula with

As consultas complexas serão muito mais fáceis de escrever e entender se forem divididas em views menores que são, então, combinadas, exatamente como estruturamos programas dividindo suas tarefas em procedimentos. Entretanto, diferente de uma definição de procedimento, uma cláusula **create view** cria uma definição de view no banco de dados, e a definição de view permanece no banco

de dados até que um comando **drop view** *nome_visão* seja executado.

A cláusula **with** fornece uma maneira de definir uma view temporária cuja definição está disponível apenas para a consulta em que a cláusula **with** ocorre. Considere a consulta a seguir, que seleciona contas com o saldo máximo; se houver muitas contas com o mesmo saldo máximo, todas elas serão selecionadas.

```
with saldo_máximo (valor) as
    select max (saldo)
    from conta
select número_conta
from conta, saldo_máximo
where conta.saldo = saldo_máximo.valor
```

A cláusula **with**, introduzida na SQL:1999, é atualmente aceita apenas por alguns bancos de dados.

Poderíamos ter escrito essa consulta usando uma subconsulta aninhada na cláusula **from** ou na cláusula **where**. Entretanto, usar subconsultas aninhadas teria tornado a consulta mais difícil de ler e de entender. A cláusula **with** torna a lógica da consulta mais clara; ela também permite que uma definição de view seja usada em vários lugares dentro de uma consulta.

Por exemplo, suponha que desejamos encontrar todas as agências em que o depósito em conta total é maior do que a média dos depósitos totais em todas as agências. Podemos escrever a consulta usando a cláusula **with** assim:

```
with total_agência (nome_agência, valor) as
    select nome_agência, sum (saldo)
    from conta
    group by nome_agência
with média_total_agência (valor) as
    select avg (valor)
    from total_agência
select nome_agência
from total_agência, média_total_agência
where total_agência.valor >= média_total_agência.valor
```

Podemos, é claro, criar uma consulta equivalente sem a cláusula **with**, mas ela seria mais complicada e mais difícil de entender. Você pode escrever a consulta equivalente como um exercício.

Views

Em nossos exemplos até este ponto, operamos no nível de modelo lógico. Ou seja, consideramos que as relações na coleção que recebemos são as relações reais armazenadas no banco de dados.

Não é desejável que todos os usuários vejam o modelo lógico inteiro. As considerações de segurança podem exigir que certos dados sejam ocultos dos usuários. Considere uma pessoa que precisa saber o número de empréstimo e o nome da agência de um cliente, mas não precisa ver o valor do empréstimo. Essa pessoa deve ver uma relação descrita (renomeação de módulo dos atributos) na SQL, por

```
select nome_cliente, tomador.número_empréstimo, nome_
       agência
from tomador, empréstimo
where tomador.número_empréstimo = empréstimo.número_
       empréstimo
```

Questões de segurança à parte, podemos querer criar uma coleção personalizada de relações que corresponda melhor à intuição do usuário do que o modelo lógico. Um funcionário do departamento de publicidade, por exemplo, poderia querer ver uma relação consistindo nos clientes que possuem uma conta ou um empréstimo no banco, além das agências com as quais eles fazem negócios. A relação para esse funcionário seria

```
(select nome_agência, nome_cliente
from depositante, conta
where depositante.número_conta = conta.número_conta)
        union
(select nome_agência, nome_cliente
from tomador, empréstimo
where tomador.número_empréstimo =
empréstimo.número_empréstimo
```

Qualquer relação que não seja parte do modelo lógico, mas que se torna visível a um usuário como uma relação virtual, é chamada de **view**. É possível aceitar um grande número de views no topo de qualquer conjunto de relações reais.

Definição de view

Definimos uma view na SQL usando o comando **create view**. Para isso, é preciso atribuir-lhe um nome e formular a consulta que calcula a view. A forma do comando **create view** é

create view *v* **as** <expressão de consulta>

onde <expressão de consulta> é qualquer expressão de consulta válida. O nome da view é representado por *v*.

Como um exemplo, considere a view consistindo nas agências e seus clientes. Suponha que queremos que essa view seja chamada de *todos_clientes*. Definimos essa view da seguinte maneira:

```
create view todos_clientes as
   (select nome_agência, nome_cliente
   from depositante, conta
   where depositante.número_conta = conta.número_conta)
union
   (select nome_agência, nome_cliente
   from tomador, empréstimo
   where tomador.número_empréstimo =empréstimo.
      número_empréstimo
```

Uma vez que definimos uma view, podemos usar o nome da view para se referir à relação virtual gerada por ela. Usando a view *todos_clientes*, podemos encontrar todos os clientes da agência Perryridge escrevendo

```
select nome_cliente
from todos_clientes
where nome_agência = 'Perryridge'
```

Os nomes de view podem aparecer em qualquer lugar onde um nome de relação pode aparecer, desde que nenhuma operação de atualização seja executada nas views. Estudamos o problema das operações de atualização nas views na seção "Atualização de uma view".

Os nomes de atributo de uma view podem ser especificados explicitamente como:

```
create view empréstimo_total_agência (nome_agência,
      empréstimo_total) as
select nome_agência, sum (quantia)
from empréstimo
group by nome_agência
```

A view anterior fornece para cada agência a soma das quantias de todos os empréstimos na agência. Como a expressão **sum** (*quantia*) não possui um nome, o nome de atributo é especificado explicitamente na definição de view.

Intuitivamente, em qualquer dado momento, o conjunto de tuplas na relação de view é o resultado da avaliação da expressão de consulta que define a view nesse momento. Portanto, se uma relação de view é calculada e armazenada, ela pode se tornar desatualizada se as relações usadas para defini-la forem modificadas. Para evitar isso, as views normalmente são implementadas da seguinte maneira. Quando definimos uma view, o sistema de banco de dados armazena a definição da view propriamente dita, em vez do resultado da avaliação da expressão de álgebra relacional que define a view. Onde quer que a relação de view apareça em uma consulta, ela é substituída pela expressão de consulta armazenada. Portanto, sempre que avaliamos a consulta, a relação de view é recalculada.

Certos sistemas de banco de dados permitem armazenar relações de view, mas eles se certificam de que, se as relações reais usadas na definição de view mudarem, a view seja mantida atualizada. Essas views são chamadas de **views materializadas**. O processo de manter a view atualizada é chamado de **manutenção de view**, discutido na seção "Views materializadas" do Capítulo 14. As aplicações que usam uma view frequentemente se beneficiam do uso das views materializadas, assim como as aplicações que exigem resposta rápida a certas consultas baseadas em view. É claro, as vantagens da materialização de uma view para as consultas precisam ser pesadas juntamente com os custos de armazenamento e o overhead adicional das atualizações.

Views definidas usando outras views

Na seção anterior, mencionamos que as relações de view podem aparecer em qualquer lugar que um nome de relação pode aparecer, exceto para restrições no uso de views em expressões de atualização. Portanto, uma view pode ser usada na expressão que define outra view. Por exemplo, podemos definir a view *cliente_perryridge* desta maneira:

```
create view cliente_perryridge as
select nome_cliente
from todos_clientes
where nome_agência = 'Perryridge'
```

onde *todos_clientes* é, ele próprio, uma relação de view.

A **expansão de view** é uma forma de definir o significado das views em termos de outras views. O procedimento considera que as definições de view não são **recursivas**; ou seja, nenhuma view é usada em sua própria definição, quer diretamente ou indiretamente por outras definições de view. Por exemplo, se $v1$ é usada na definição de $v2$, $v2$ é usada na definição de $v3$ e $v3$ é usada na definição de $v1$, então, $v1$, $v2$ e $v3$ são recursivas. As definições de view recursivas são úteis em algumas situações, e as veremos novamente no contexto da linguagem Datalog, na seção "Datalog" do Capítulo 5.

Façamos a view v_1 ser definida por uma expressão e_1 que pode, ela própria, conter usos das relações de view. Uma relação de view representa a expressão definindo a view e, portanto, uma relação de view pode ser substituída pela expressão que a define. Se modificarmos uma expressão substituindo uma relação de view pela definição posterior, a expressão resultante ainda pode conter outras relações de view. Conseqüentemente, a expansão de view de uma expressão repete o passo de substituição desta forma:

```
repeat
   Encontre qualquer relação de view v_i em e_1
   Substitua a relação de view v_i pela expressão que define v_i
   until nenhuma outra relação de view esteja presente em e_1
```

Desde que as definições de view não sejam recursivas, esse loop terminará. Portanto, uma expressão e contendo relações de view pode ser entendida como a expressão resultante da expansão de view de e, que não contém quaisquer relações de view.

Como uma ilustração da expansão de view, considere a seguinte expressão:

```
select *
from cliente_perryridge
where nome_cliente = 'John'
```

O procedimento de expansão de view inicialmente gera

```
select *
from (select nome_cliente
      from todos_clientes
      where nome_agência = 'Perryridge'
     where nome_cliente = 'John'
```

Ela, então, gera

```
select *
from (select nome_cliente
   from ((select nome_agência, nome_cliente
        from depositante, conta
        where depositante.número_conta =
conta.número_conta)
        union
       (select nome_agência, nome_cliente
        from tomador, empréstimo
        where tomador.número_empréstimo = empréstimo.
            número_empréstimo))
   where nome_agência = 'Perryridge'
where nome_cliente = 'John'
```

Nesse momento, não existem mais usos de relações de view, e a expansão de view termina.

Modificação do banco de dados

Até agora, restringimos nossa atenção à extração de informações do banco de dados. Agora, mostraremos como acrescentar, remover ou mudar informações com a SQL.

Exclusão

Uma requisição de exclusão é expressa quase da mesma maneira de uma consulta. É possível excluir apenas tuplas inteiras; não podemos excluir valores apenas em atributos específicos. A SQL expressa uma exclusão por

```
delete from r
where P
```

onde P representa um predicado e r representa uma relação. A instrução **delete** primeiramente encontra todas as tuplas t em r para as quais $P(t)$ seja verdadeiro; depois, ela as exclui de r. A cláusula **where** pode ser omitida, caso em que todas as tuplas em r são excluídas.

Observe que um comando **delete** opera apenas em uma relação. Se quisermos excluir tuplas de várias relações, precisamos usar um comando **delete** para cada relação. O predicado na cláusula **where** pode ser tão complexo quanto a cláusula **where** de um comando **select**. No outro extremo, a cláusula **where** pode estar vazia. A requisição

```
delete from empréstimo
```

exclui todas as tuplas da relação *empréstimo*. (Os sistemas bem-projetados pedirão a confirmação do usuário antes de executar uma requisição tão devastadora.)

Aqui estão exemplos de requisições de exclusão SQL:

- Exclua todas as tuplas *conta* na agência Perryridge.

```
delete from conta
where nome_agência = 'Perryridge'
```

- Exclua todos os empréstimos com quantia entre $1.300 e $1.500.

```
delete from empréstimo
where quantia between 1300 and 1500
```

- Exclua todas as tuplas *conta* em cada agência localizada em Brooklyn.

```
delete from conta
where nome_agência in (select nome_agência
                       from agência
                       where cidade_agência = 'Brooklyn')
```

Essa requisição **delete** primeiro encontra todas as agências em Brooklyn e, depois, exclui todas as tuplas *conta* pertencentes a essas agências.

Note que, embora possamos excluir tuplas apenas de uma relação de cada vez, podemos referenciar qualquer número de relações em um **select-from-where** aninhado na cláusula **where** de um **delete**. A requisição **delete** pode conter um **select** aninhado que referencia a relação da qual as tuplas devem ser excluídas. Por exemplo, suponha que queiramos excluir os registros de todas as contas com saldos abaixo da média no banco. Poderíamos escrever

```
delete from conta
where saldo < (select avg (saldo)
               from conta)
```

A instrução **delete** primeiro testa cada tupla na relação *conta* para verificar se a conta possui um saldo menor que a média no banco. Em seguida, todas as tuplas que falham no teste – ou seja, que representam uma conta com um saldo menor do que a média – são excluídas. É importante realizar todos os testes antes de executar qualquer exclusão – se alguma tupla for excluída antes de outras tuplas terem sido testadas, o saldo médio poderá mudar e o resultado final do **delete** dependerá da ordem em que as tuplas foram processadas!

Inserção

Para inserir dados em uma relação, especificamos uma tupla a ser inserida ou escrevemos uma consulta cujo resultado seja um conjunto de tuplas a ser inserido. Obviamente, os valores de atributo para tuplas inseridas precisam ser membros do domínio do atributo. Da mesma forma, as tuplas inseridas precisam ser da aridade correta.

A instrução **insert** mais simples é uma requisição para inserir uma tupla. Suponha que desejamos inserir o fato de que existe uma conta A-9732 na agência Perryridge e que ela tem um saldo de $1.200. Escrevemos

```
insert into conta
    values ('A-9732', 'Perryridge', 1200)
```

Nesse exemplo, os valores são especificados na ordem em que os atributos correspondentes são listados no esquema de relação. Para o benefício dos usuários que podem não se lembrar da ordem dos atributos, a SQL permite que os atributos sejam especificados como parte da instrução **insert**. Por exemplo, as seguintes instruções **insert** SQL são idênticas em função à instrução anterior:

```
insert into conta (número_conta, nome_agência, saldo)
    values ('A-9732', 'Perryridge', 1200)

insert into conta (nome_agência, número_conta, saldo)
    values ('Perryridge', 'A-9732', 1200)
```

Mais geralmente, podemos querer inserir tuplas com base no resultado de uma consulta. Suponha que queiramos oferecer uma nova conta de poupança no valor de $200 como um presente para todos os clientes de empréstimo da agência Perryridge, para cada empréstimo que eles possuírem. Considerando que o número de empréstimo serve como o número de conta para a conta de poupança, escrevemos

```
insert into conta
    select número_empréstimo, nome_agência, 200
    from empréstimo
    where nome_agência = 'Perryridge'
```

Em vez de especificar uma tupla como fizemos anteriormente nesta seção, usamos um **select** para especificar um conjunto de tuplas. A SQL primeiramente avalia a instrução **select**, fornecendo um conjunto de tuplas que é, então, inserido na relação *conta*. Cada tupla possui um *número_empréstimo* (que serve como o número da nova conta), um *nome_agência* (Perryridge) e um saldo inicial da nova conta ($200).

Também é preciso acrescentar tuplas à relação *depositante, o que pode ser feito* escrevendo

```
insert into depositante
    select nome_cliente, número_empréstimo
    from tomador, empréstimo
    where tomador.número_empréstimo = empréstimo.
        número_empréstimo and
        nome_agência = 'Perryridge'
```

Essa consulta insere uma tupla (*nome_cliente, número_empréstimo*) na relação *depositante* para cada *nome_cliente* que tem um empréstimo na agência Perryridge com número de empréstimo *número_empréstimo*.

É importante que avaliemos a instrução **select** completamente antes de realizar quaisquer inserções. Se efetuarmos alguma inserção mesmo enquanto a instrução **select** está sendo avaliada, uma requisição como

```
insert into conta
    select *
    from conta
```

poderia inserir um número infinito de tuplas! A requisição inseriria a primeira tupla em *conta* novamente, criando uma segunda cópia da tupla. Como essa segunda cópia agora é parte de *conta*, a instrução **select** pode encontrá-la, e uma terceira cópia seria inserida em *conta*. A instrução **select**, então, pode encontrar essa terceira cópia e inserir uma quarta cópia e assim por diante, interminavelmente. Avaliar completamente a instrução **select** antes de realizar inserções evita esses problemas.

Nossa discussão da instrução **insert** considerou apenas exemplos em que um valor é dado para cada atributo nas tuplas inseridas. É possível, como vimos no Capítulo 2, que as tuplas inseridas recebam valores em apenas alguns atributos do esquema. Os atributos restantes recebem a atribuição de um valor nulo indicado por *null*. Considere a requisição

```
insert into conta values('A-401',null,1200)
```

Sabemos que a conta A-401 possui $1200, mas o nome da agência não é conhecido. Considere a consulta

```
select numero_conta
from conta
where nome_agência = 'Perryridge'
```

Como a agência em que a conta A-401 é mantida não é conhecida, não podemos determinar se ela é igual a "Perryridge".

Podemos proibir a inserção de valores nulos em atributos especificados usando a DDL SQL, que discutimos na seção "Definição de dados".

A maioria dos produtos de bancos de dados relacionais possui utilitários de "bulk loader" para inserir uma grande quantidade de tuplas em uma relação. Esses utilitários permitem que dados sejam lidos de arquivos de texto formatados e possam ser executados muito mais rápido do que uma seqüência de instruções **insert** equivalentes.

Atualizações

Em certas situações, podemos desejar mudar um valor em uma tupla sem mudar *todos* os valores na tupla. A instrução **update** pode ser usada nesse caso. Assim como para **insert** e **delete**, podemos escolher as tuplas a serem atualizadas usando uma consulta.

Suponha que estejam sendo feitos pagamentos de juros anuais, e todos os saldos devem ser aumentados em 5%. Escrevemos

```
update conta
set saldo = saldo * 1.05
```

Essa instrução update é aplicada apenas uma vez em cada uma das tuplas na relação *conta*.

Se os juros devem ser pagos apenas às contas com um saldo de $1.000 ou mais, escrevemos

```
update conta
set saldo = saldo * 1.05
where saldo >= 1000
```

Em geral, a cláusula **where** da instrução **update** pode conter qualquer construção válida na cláusula **where** da instrução **select** (incluindo **selects** aninhados). Assim como em **insert** e **delete**, um **select** aninhado dentro de uma instrução **update** pode referenciar a relação que está sendo atualizada. Como antes, a SQL primeiro testa todas as tuplas na relação para ver se elas devem ser atualizadas e realiza as atualizações após isso. Por exemplo, podemos escrever a requisição "Pague 5% de juros às contas cujo saldo seja maior do que a média" desta forma:

```
update conta
set saldo = saldo * 1.05
where saldo > (select avg (saldo)
               from conta)
```

Agora, vamos supor que todas as contas com saldos acima de $10.000 recebam 6% de juros, enquanto todas as outras recebem 5%. Poderíamos escrever duas instruções **update**:

```
update conta
set saldo = saldo * 1.06
where saldo > 10000

update conta
set saldo = saldo * 1.05
where saldo <= 10000
```

Repare que, como vimos no Capítulo 2, a ordem das duas instruções **update** é importante. Se mudássemos a ordem das duas instruções, uma conta com um saldo logo abaixo de $10.000 receberia 11,3% de juros.

A SQL fornece uma construção **case**, que podemos usar para realizar as duas atualizações com uma única instrução **update**, evitando o problema com a ordem das atualizações.

```
update conta
set saldo = case
        when saldo <= 10000 then saldo * 1.05
        else saldo * 1.06
    end
```

A forma geral da instrução **case** é a seguinte:

```
case
        when pred1 then resultado1
        when pred2 then resultado2
        ...
        when predn then resultadon
        else resultado0
end
```

A operação retorna $resultado_i$, onde i é o primeiro dos predicados $pred_1, pred_2, \ldots, pred_n$ que é satisfeito; se nenhum dos predicados for satisfeito, a operação retorna $resultado_0$. As instruções case podem ser usadas em qualquer lugar onde um valor é esperado.

Atualização de uma view

Embora sejam uma ferramenta útil para as consultas, as views apresentam sérios problemas se usadas para expressar atualizações, inserções ou exclusões. A dificuldade é que uma modificação no banco de dados expressa em termos de uma view precisa ser traduzida para uma modificação nas relações reais no modelo lógico do banco de dados.

Para ilustrar o problema, considere um escriturário que precisa ver todos os dados de empréstimo na relação *em-*

préstimo, exceto *quantia_empréstimo*. Vamos fazer com que *agência_empréstimo* seja a view fornecida para o escriturário. Definimos essa view como

create view *agência_empréstimo* **as**
select *número_empréstimo, nome_agência*
from *empréstimo*

Como permitimos que um nome de view apareça em qualquer lugar onde um nome de relação é permitido, o escriturário pode escrever:

insert ingo *agência_empréstimo*
values ('L-37', 'Perryridge')

Essa inserção precisa ser representada por uma inserção na relação *empréstimo*, já que *empréstimo* é a real relação da qual o sistema de banco de dados constrói a view *agência_empréstimo*. Entretanto, para inserir uma tupla em *empréstimo*, precisamos ter algum valor para *quantia*. Existem dois métodos satisfatórios para lidar com essa inserção:

- Rejeitar a inserção e retornar uma mensagem de erro para o usuário.
- Inserir uma tupla (L-37, "Perryridge", *nulo*) na relação *empréstimo*.

Outro problema com a modificação do banco de dados por meio de views ocorre com uma view do tipo

create view *info_empréstimo* **as**
select *nome_cliente, quantia*
from *tomador, empréstimo*
where *tomador.número_empréstimo = empréstimo.número_empréstimo*

Essa view lista a quantia para cada **empréstimo** que qualquer cliente do banco possui. Considere a seguinte inserção por meio dessa view:

insert into *info_empréstimo*
values ('Johnson', 1900)

O único método possível de inserir tuplas nas relações *tomador* e *empréstimo* é inserir ("Johnson", *nulo*) em *tomador* e (*nulo*, *nulo*, 1900) em *empréstimo*. Depois, obtemos as relações mostradas na Figura 3.3. Entretanto, essa atualização não possui o efeito desejado, já que a relação de view *info_empréstimo* ainda *não* inclui a tupla ("Johnson", 1900). Portanto, não há uma maneira de atualizar as relações *tomador* e *empréstimo* usando nulos para obter a atualização desejada em *info_empréstimo*.

Devido a problemas como esses, as modificações geralmente não são permitidas nas relações de view, exceto em casos limitados. Diferentes sistemas de banco de dados especificam diferentes condições sob as quais permitem atualizações nas relações de view; veja o manual do sistema de banco de dados para obter detalhes. O problema geral da modificação de banco de dados por meio de views tem sido objeto de muita pesquisa, e as notas bibliográficas fornecem indicações para algumas dessas pesquisas.

Em geral, uma view SQL é chamada de **atualizáveis** (ou seja, inserções, atualizações ou exclusões podem ser aplicadas na view) se todas as condições a seguir forem satisfeitas:

- A cláusula **from** possui apenas uma relação de banco de dados.
- A cláusula **select** contém apenas nomes de atributo da relação e não possui quaisquer expressões, agregadas ou especificação **distinct**.
- Qualquer atributo não listado na cláusula **select** pode ser definido em nulo.
- A consulta não possui uma cláusula **group by** ou **having**.

número_empréstimo	*nome_agência*	*quantia*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500
null	null	1900

loan

nome_cliente	*número_empréstimo*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	null

borrower

Figura 3.3 Tuplas inseridas em *empréstimo* e *tomador*.

Sob essas restrições, as operações **update**, **insert** e **delete** seriam proibidas na view de exemplo *todos_clientes* que definimos anteriormente.

Suponha que uma view *conta_downtown* seja definida desta forma:

> **create view** *conta_downtown* **as**
> **select** *numero_conta, nome_agência, saldo*
> **from** *conta*
> **where** *nome_agência* = 'Downtown'

Essa view é atualizável, já que ela satisfaz as condições listadas anteriormente.

Mesmo com as condições sobre atualização, o seguinte problema ainda permanece. Suponha que um usuário tente inserir a tupla ('A-999', 'Perryridge', 1000) na view *conta_downtown*. Essa tupla pode ser inserida na relação *conta*, mas ela não apareceria na view *conta_downtown*, já que não satisfaz a seleção imposta pela view.

Como padrão, a SQL permitiria essa atualização. Entretanto, as views podem ser definidas com uma cláusula **with check option** no final da definição de view; então, se uma tupla inserida na view não satisfizer a condição da cláusula **where** da view, a inserção é rejeitada pelo sistema de banco de dados. As atualizações são semelhantemente rejeitadas se o novo valor não satisfizer as condições da cláusula **where**.

A SQL:1999 possui um conjunto de regras mais complexo sobre quando inserções, atualizações e exclusões podem ser executadas em uma view, que permite atualizações por meio de uma classe maior de views; no entanto, as regras são muito complexas para serem discutidas aqui.

Transações

Uma **transação** consiste em uma seqüência de instruções de consulta e/ou atualização. O padrão SQL especifica que uma transação inicia implicitamente quando uma instrução SQL é executada. Uma das seguintes instruções SQL precisam finalizar a transação:

- **Commit work** confirma a transação atual; ou seja, torna as atualizações realizadas pela transação permanentes no banco de dados. Após a transação ser confirmada, uma nova transação é automaticamente iniciada.
- **Rollback work** faz com que a transação atual seja revertida; ou seja, ele desfaz todas as atualizações realizadas pelas instruções SQL na transação. Portanto, o estado do banco de dados é restaurado para como era antes de a primeira instrução da transação ser executada.

A palavra-chave **work** é opcional nas duas instruções.
A transação rollback é útil se alguma condição de erro for detectada durante a execução da transação. Commit é semelhante, de certo modo, a salvar mudanças em um documento que está sendo editado, enquanto rollback é semelhante a sair da seção de edição sem salvar as mudanças. Uma vez que a transação tenha executado **commit work**, seus efeitos não podem mais ser desfeitos por **rollback work**. O sistema de banco de dados garante que, no caso de alguma falha, como um erro em uma das instruções SQL, uma interrupção de energia ou uma falha do sistema, os efeitos de uma transação serão revertidos se ele ainda não tiver executado o **commit work**. No caso de uma interrupção de energia ou falha do sistema, o rollback ocorre quando o sistema reinicia.

Por exemplo, para transferir dinheiro de uma conta para outra, precisamos atualizar dois saldos de conta. As duas instruções de atualização formariam uma transação. Um erro enquanto uma transação executa uma de suas instruções resultaria em desfazer os efeitos das instruções anteriores da transação, de modo que o banco de dados não seja deixado em um estado parcialmente atualizado. Veremos mais propriedades das transações no Capítulo 15.

Se um programa terminar sem executar um desses comandos, as atualizações são confirmadas ou revertidas. O padrão não especifica qual dos dois ocorre, e a escolha depende da implementação. Em muitas implementações SQL, por padrão, cada instrução SQL é tomada para ser uma transação em si mesma e é confirmada assim que é executada. Um commit automático de instruções SQL individuais precisa ser desativado se uma transação consistindo em múltiplas instruções SQL precisar ser executada. Como desativar um commit automático depende da implementação SQL específica.

Uma alternativa melhor, que é parte do padrão SQL:1999 (mas aceito por apenas algumas implementações SQL atualmente), é permitir que múltiplas instruções SQL sejam colocadas entre as palavras-chave **begin atomic** ... **end**. Todas as instruções entre as palavras-chave, então, formam uma única transação.

Relações juntadas**

A SQL fornece não só o mecanismo de produto cartesiano básico para juntar tuplas das relações, mas também fornece (nas versões SQL-92 e posteriores) vários outros mecanismos para juntar relações, incluindo junções de condição e junções naturais, bem como várias formas de junções externas. Essas operações adicionais normalmente são usadas como expressões de subconsulta na cláusula **from**.

Exemplos

Ilustramos as várias operações de junção usando as relações *empréstimo* e *tomador* na Figura 3.4. Começamos com

número_empréstimo	nome_agência	quantia
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	17000

nome_cliente	número_empréstimo
Jones	L-170
Smith	L-230
Hayes	L-155

Figura 3.4 Relações *empréstimo* e *tomador*.

um exemplo simples de junções internas. A Figura 3.5 mostra o resultado da expressão

empréstimo **inner join** *tomador* **on**
empréstimo.número_empréstimo =
tomador.número_empréstimo

A expressão calcula a junção teta das relações *empréstimo* e *tomador*, com a condição de junção sendo *empréstimo.número_empréstimo* = *tomador.número_empréstimo*. Os atributos do resultado consistem nos atributos da relação do lado esquerdo seguidos dos atributos da relação do lado direito.

Observe que o atributo *número_empréstimo* aparece duas vezes na figura – a primeira ocorrência é de *empréstimo*, e a segunda é de *tomador*. O padrão SQL não exige que os nomes de atributo nesses resultados sejam únicos. Uma cláusula **as** deve ser usada para atribuir nomes únicos aos atributos nos resultados de consulta e subconsulta.

Renomeamos a relação de resultado de uma junção e os atributos da relação de resultado usando uma cláusula as, como ilustrado aqui:

empréstimo **inner join** *tomador* on *empréstimo.número_empréstimo* = *tomador.número_empréstimo* as *lb*(*número_empréstimo, agência, quantia, custo, num_empréstimo_custo*)

Renomeamos a segunda ocorrência de *número_empréstimo* para *num_empréstimo_custo*. A ordem dos atributos no resultado da junção é importante para a renomeação.

A seguir, consideramos um exemplo da operação left outer-join.

empréstimo **left outer join** *tomador* **on**
empréstimo.número_empréstimo =
tomador.número_empréstimo

Podemos calcular logicamente a operação de junção externa esquerda da seguinte maneira. Primeiro, calculamos o resultado da junção interna como antes. Depois, para cada tupla *t* na relação *empréstimo* do lado esquerdo que não corresponde a qualquer tupla na relação *tomador* do lado direito na junção interna, acrescente uma tupla *r* ao resultado da junção: os atributos da tupla *r* que são derivados da relação do lado esquerdo são preenchidos com os valores da tupla *t*, e os atributos restantes de *r* são preenchidos com valores nulos. A Figura 3.6 mostra a relação resultante. As tuplas (L-170, Downtown, 3000) e (L-230, Redwood, 4000) se juntam com tuplas de *tomador* e aparecem no resultado da junção interna, e, portanto, no resultado da junção externa esquerda. Por outro lado, a tupla (L-260, Perryridge, 1700) não corresponde a qualquer tupla de *tomador* na junção interna, e, portanto, uma tupla (L-260, Perryridge, 1700, nulo, nulo) está presente no resultado da junção externa esquerda.

número_empréstimo	nome_agência	quantia	nome_cliente	número_empréstimo
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

Figura 3.5 Resultado de *empréstimo* **inner join** *tomador* **on** *empréstimo.número_empréstimo = tomador.número_empréstimo*.

número_empréstimo	nome_agência	quantia	nome_cliente	número_empréstimo
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*nulo*	*nulo*

Figura 3.6 Resultado de *empréstimo* **left outer join** *tomador* **on** *empréstimo.número_empréstimo = tomador.número_empréstimo*.

número_empréstimo	*nome_agência*	*quantia*	*nome_cliente*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Figura 3.7 Resultado de *empréstimo* **natural inner join** *tomador*.

Finalmente, consideramos um exemplo da operação natural-join:

empréstimo **natural inner join** *tomador*

Essa expressão calcula a junção natural das duas relações. O único nome de atributo comum a *empréstimo* e *tomador* é *número_empréstimo*. A Figura 3.7 mostra o resultado da expressão. O resultado é semelhante ao resultado da junção interna com a condição **on** na Figura 3.5, já que eles possuem, na verdade, a mesma condição de junção. Entretanto, o atributo *número_empréstimo* aparece apenas uma vez no resultado da junção natural, enquanto ele aparece duas vezes no resultado da junção com a condição **on**.

Tipos e condições de junção

Na seção anterior, vimos exemplos das operações de junção permitidas na SQL. As operações de junção tomam duas relações e retornam outra relação como resultado. Embora as expressões de junção externa normalmente sejam usadas na cláusula **from**, elas podem ser usadas no mesmo lugar que uma relação.

Cada uma das variantes das operações de junção na SQL consiste em um *tipo de junção* e uma *condição de junção*. A condição de junção define que tuplas nas duas relações são correspondentes e que atributos estão presentes no resultado da junção. O tipo de junção define como são tratadas as tuplas em cada relação que não correspondem a qualquer tupla na outra relação (com base na condição de junção). A Figura 3.8 mostra alguns dos tipos e condições de junção permitidos. O primeiro tipo de junção é a junção interna e os outros três são as junções externas.

Das três condições de junção, já vimos a junção **natural** e a condição **on**, e discutiremos a condição **using** mais adiante nesta seção.

O uso de uma condição de junção é obrigatório para as junções externas, mas é opcional para as internas (se ela for omitida, um produto cartesiano resulta). Sintaticamente, a palavra-chave **natural** aparece antes do tipo de junção, como ilustrado anteriormente, enquanto as condições **on** e **using** aparecem no final da expressão de junção. As palavras-chaves **inner** e **outer** são opcionais, já que o restante do tipo de junção nos permite deduzir se a junção é interna ou externa.

O significado da condição de junção **natural**, em termos de quais tuplas das duas relações correspondem, é simples. A ordem dos atributos no resultado de uma junção natural é a seguinte. Os atributos de junção (isto é, os atributos comuns às duas relações) aparecem primeiro, na ordem em que aparecem na relação do lado esquerdo. Em seguida, vêm todos os atributos não-junção da relação do lado esquerdo, e, finalmente, todos os atributos não-junção da relação do lado direito.

A junção externa direita é simétrica à junção externa esquerda. As tuplas da relação do lado direito que não correspondem a qualquer tupla na relação do lado esquerdo são preenchidas com nulos e são acrescentadas ao resultado da junção externa direita.

Aqui está um exemplo de combinar a condição de junção natural com o tipo de junção externa direita:

empréstimo **natural right outer join** *tomador*

A Figura 3.9 mostra o resultado dessa expressão. Os atributos do resultado são definidos pelo tipo de junção, que é uma junção natural; por isso, *número_empréstimo*

Tipos de junção
Junção interna (**inner join**)
Junção externa esquerda (**left outer join**)
Junção externa direita (**right outer join**)
Junção externa completa (**full outer join**)

Condições de junção
Natural (**natural**)
on <predicado>
using (A_1, A_2,..., A_n)

Figura 3.8 Tipos de junção e condições de junção.

número_empréstimo	*nome_agência*	*quantia*	*nome_cliente*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*Nulo*	*nulo*	Hayes

Figura 3.9 Resultado de *empréstimo* **natural right outer join** *tomador*.

aparece apenas uma vez. As duas primeiras tuplas no resultado são da junção natural interna de *empréstimo* e *tomador*. A tupla (Hayes, L-155) da relação do lado direito não corresponde a qualquer tupla da relação *empréstimo* do lado esquerdo na junção interna natural. Consequentemente, a tupla (L-155, nulo, nulo, Hayes) aparece no resultado da junção.

A condição de junção **using** $(A_1, A_2, ..., A_n)$ é semelhante à condição de junção natural, exceto que os atributos de junção são os atributos $A_1, A_2, ..., A_n$ em vez de todos os atributos que são comuns às duas relações. Os atributos $A_1, A_2, ..., A_n$ precisam consistir apenas em atributos que são comuns às duas relações, e eles aparecem apenas uma vez no resultado da junção.

A junção externa completa é uma combinação dos tipos de junção externa esquerda e direita. Após a operação calcular o resultado da junção interna, ela se estende com tuplas nulas da relação do lado esquerdo que não corresponderam a qualquer tupla do lado direito e, depois, as acrescenta ao resultado. Da mesma forma, ela se estende com tuplas nulas da relação do lado direito que não corresponderam a qualquer tupla do lado esquerdo e as acrescenta ao resultado.

Por exemplo, a Figura 3.10 mostra o resultado da expressão

empréstimo **full outer join** *tomador* **using** (*número_empréstimo*)

Como outro exemplo do uso da operação de junção externa, podemos escrever a consulta "Encontre todos os clientes que possuem uma conta mas não um empréstimo no banco" como

```
select d_CN
from (depositante left outer join tomador
    on depositante.nome_cliente = tomador.nome_cliente)
    as bd1 (d_CN, número_conta, b_CN,
número_empréstimo)
where b_CN is null
```

Semelhantemente, podemos escrever a consulta "Encontre todos os clientes que possuem uma conta ou um empréstimo (mas não ambos) no banco" com junções externas naturais completas como:

```
select nome_cliente
from (depositante natural full outer join tomador)
where número_conta is null or número_empréstimo is null
```

A SQL-92 também fornece dois outros tipos de junção, chamadas **junção cruzada (cross join)** e **junção união** (union join). A primeira é equivalente a uma junção interna sem uma condição de junção; a segunda é equivalente a uma junção externa completa na condição "false" – ou seja, onde a junção interna está vazia.

Resumo

- Os sistemas de banco de dados comerciais não usam a álgebra relacional concisa e formal abordadas no Capítulo 2. A linguagem SQL amplamente usada, que estudamos neste capítulo, é baseada na álgebra relacional, mas inclui muito "açúcar sintático".
- A linguagem de definição de dados da SQL é usada para criar relações com esquemas especificados. A DDL SQL aceita diversos tipos, incluindo **date e time**. Mais detalhes

número_empréstimo	*nome_agência*	*quantia*	*nome_cliente*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*nulo*
L-155	*Nulo*	*nulo*	Hayes

Figura 3.10 Resultado de *empréstimo* **full outer join** *tomador* **using** (*número_empréstimo*).

sobre a DDL SQL, especialmente seu suporte a restrições de integridade, aparecem na seção "Definição de dados".

- A SQL inclui uma variedade de construções de linguagem para consultas no banco de dados. Todas as operações de álgebra relacional, incluindo as operações de álgebra relacional estendidas, podem ser expressas pela SQL. A SQL também permite a ordenação dos resultados de consulta classificando em atributos especificados.
- A SQL manipula consultas em relações contendo valores nulos acrescentando o valor verdade "desconhecido" aos valores verdade comuns de true e false.
- A SQL permite subconsultas aninhadas na cláusula where. A consulta externa pode realizar várias operações no resultado da subconsulta, como verificar a ausência ou presença de um valor no resultado da subconsulta. As subconsultas na cláusula from são chamadas de relações derivadas.
- As relações de view podem ser definidas como relações contendo o resultado das consultas. As views são úteis para ocultar informações desnecessárias e para reunir informações de mais de uma relação em uma única view.
- As views temporárias definidas usando a cláusula **with** também são úteis para desmembrar consultas complexas em partes menores e mais fáceis de entender.
- A SQL fornece construções para atualizar, inserir e excluir informações. As atualizações por meio de views só são permitidas quando algumas condições bastante restritivas são satisfeitas.
- As transações são uma conseqüência das consultas e atualizações que, juntas, realizam uma tarefa. As transações podem ser confirmadas ou revertidas; quando uma transação é revertida, os efeitos de todas as atualizações realizadas pela transação são desfeitos.
- A SQL aceita vários tipos de junção externa, com vários tipos de condições de junção.

Termos de revisão

- Linguagem de definição de dados (DDL)
- Linguagem de manipulação de dados (DML)
- Cláusula **select**
- Cláusula **from**
- Cláusula **where**
- Cláusula **as**
- Variável de tupla
- Cláusula **order by**
- Duplicatas
- Operações de conjunto
  - **union**, **intersect**, **except**
- Funções agregadas
  - **avg**, **min**, **max**, **sum**, **count**, **group by**
- Valores nulos
  - Valor verdade "desconhecido"
- Subconsultas aninhadas
- Operações de conjunto
  - {<, <=, >, >=} {**some**, **all**}
  - **exists**
  - **unique**
- Relações derivadas (na cláusula **from**)
- Cláusula **with**
- Views
  - Definição de view
  - Expansão de view
- Modificação de banco de dados
  - **delete**, **insert**, **update**
  - Atualização de view
- Transação
  - commit
  - rollback
- Tipos de junção
  - junção interna e externa
  - junção externa esquerda, direita e completa
  - natural, using e on

Exercícios práticos

3.1 Considere o banco de dados de seguros da Figura 3.11, no qual as chaves primárias estão sublinhadas. Construa as seguintes consultas SQL para esse banco de dados relacional.

a. Encontre o número total de pessoas que possuíam carros que foram envolvidos em acidentes em 1989.

b. Inclua um novo acidente no banco de dados; considere quaisquer valores para os atributos necessários.

c. Exclua o Mazda pertencente a "John Smith".

pessoa (*id_motorista*, *nome*, *endereço*)
carro (*licença*, *modelo*, *ano*)
acidente (*número_ocorrência*, *data*, *local*)
pertence (*id_motorista*, *licença*)
participou (*id_motorista*, *carro*, *número_ocorrência*, *valor_dano*)

Figura 3.11 Banco de dados de seguros.

funcionário (*nome_funcionário*, *rua*, *cidade*)
trabalha (*nome_funcionário*, *nome_empresa*, *salário*)
empresa (*nome_empresa*, *cidade*)
gerencia (*nome_funcionário*, *nome_gerente*)

Figura 3.12 Banco de dados de funcionários.

3.2 Considere o banco de dados de funcionários da Figura 3.12, no qual as chaves primárias estão sublinhadas. Forneça uma expressão em SQL para cada uma das consultas a seguir.

a. Encontre os nomes e cidades de todos os funcionários que trabalham para o First Bank Corporation.
b. Encontre os nomes, endereços de rua e cidades de todos os funcionários que trabalham para o First Bank Corporation e ganham mais de $10.000.
c. Encontre todos os funcionários no banco de dados que não trabalham para o First Bank Corporation.
d. Encontre todos os funcionários no banco de dados que ganham mais do que os funcionários do Small Bank Corporation.
e. Considere que as empresas podem estar localizadas em várias cidades. Encontre todas as empresas que estão localizadas em cada cidade onde o Small Bank Corporation está localizado.
f. Encontre a empresa que possui o maior número de funcionários.
g. Encontre as empresas cujos funcionários ganham um salário médio maior do que o salário médio no First Bank Corporation.

3.3 Considere o banco de dados relacional da Figura 3.12. Forneça uma expressão em SQL para cada uma das seguintes consultas.

a. Modifique o banco de dados de modo que Jones agora resida em Newtown.
b. Dê a todos os gerentes do First Bank Corporation um aumento salarial de 10%, a menos que o salário se torne maior do que $100.000; nesses casos, dê um aumento de apenas 3%.

3.4 A SQL-92 fornece uma operação exótica chamada **coalesce**, que é definida desta forma: coalesce (A_1, A_2,..., A_n) retorna o primeiro A_i não nulo na lista A_1, A_2,..., A_n, e retorna nulo se todos de A_1, A_2,..., A_n forem nulos.

Sejam *a* e *b* relações com os esquemas *A* (*nome*, *endereço*, *cargo*) e *B* (*nome*, *endereço*, *salário*), respectivamente. Mostre como expressar *a* natural full outer join *b* usando a operação **full outer-join** com uma condição **on** e a operação **coalesce**. Certifique-se de que a relação resultado não contenha duas cópias dos atributos *nome* e *endereço*, e que a solução esteja correta mesmo se algumas tuplas em *a* e *b* tenham valores nulos para os atributos *nome* ou *endereço*.

3.5 Suponha que temos uma relação *marcas* (*id_aluno*, *pontuação*) e desejamos atribuir notas aos alunos com base na pontuação, da seguinte maneira: nota F se *pontuação* < 40; nota C se 40 ≤ *pontuação* < 60; nota B se 60 ≤ *pontuação* < 80; e nota A se 80 ≤ *pontuação*. Escreva consultas SQL para fazer o seguinte:

a. Exiba a nota para cada aluno, com base na relação *marcas*.
b. Encontre o número de alunos com cada nota.

3.6 Considere a consulta SQL

```
select p.a1
from p, r1, r2
where p.a1 = r1.a1 or p.a1 = r2.a1
```

Sob que condições a consulta anterior seleciona valores de *p.a1* que estejam em *r1* ou em *r2*? Examine cuidadosamente os casos em que *r1* ou *r2* possa estar vazio.

3.7 Certos sistemas permitem nulos *marcados*. Um nulo marcado $\perp_i$ é igual a ele mesmo, mas se $i \neq j$, então, $\perp_i \neq \perp_j$. Uma aplicação dos nulos marcados é permitir certas atualizações por meio de views. Considere a view *info_empréstimo* (seção "Views"). Mostre como você pode usar nulos marcados para permitir a inserção da tupla ("Johnson", 1900) por *info_empréstimo*.

Exercícios

3.8 Considere o banco de dados de seguros da Figura 3.11, no qual as chaves primárias estão sublinhadas. Construa as seguintes consultas SQL para esse banco de dados relacional.

a. Encontre o número de acidentes em que os carros pertencentes a "John Smith" estavam envolvidos.

b. Atualize para $3.000 o valor do dano para o carro com número de licença "AABB2000" no acidente com número de ocorrência "AR2197".

3.9 Considere o banco de dados de funcionários da Figura 3.12, no qual as chaves primárias estão sublinhadas. Dê uma expressão SQL para cada uma das seguintes consultas.

a. Encontre os nomes de todos os funcionários que trabalham para o First Bank Corporation.
b. Encontre todos os funcionários no banco de dados que moram nas mesmas cidades das empresas para as quais eles trabalham.
c. Encontre todos os funcionários no banco de dados que moram nas mesmas cidades e nas mesmas ruas em que moram seus gerentes.
d. Encontre todos os funcionários que ganham mais do que a média dos salários de todos os funcionários de sua empresa.
e. Encontre a empresa que possui a menor folha de pagamento.

3.10 Considere o banco de dados relacional da Figura 3.12. Forneça uma expressão em SQL para cada uma das seguintes consultas.

a. Dê um aumento de 10% a todos os funcionários do First Bank Corporation.
b. Dê um aumento de 10% a todos os gerentes do First Bank Corporation.
c. Exclua todas as tuplas na relação *trabalha* para funcionários do Small Bank Corporation.

3.11 Considere os seguintes esquemas de relação:

$$R = (A, B, C)$$
$$S = (D, E, F)$$

Dadas as relações $r(R)$ e $s(S)$, forneça uma expressão em SQL que seja equivalente a cada uma das seguintes consultas:

a. $\Pi_A(r)$
b. $\sigma_{B=17}(r)$
c. $r \times s$
d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

3.12 Considere $R = (A, B, C)$ e sejam r_1 e r_2 relações no esquema R. Dê uma expressão em SQL que seja equivalente a cada uma das seguintes consultas.

a. $r_1 \cup r_2$
b. $r_1 \cap r_2$
c. $r_1 - r_2$
d. $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

3.13 Mostre que, na SQL, **< > all** é idêntico a **not in**.

3.14 Considere o banco de dados relacional da Figura 3.12. Usando SQL, defina uma view consistindo em *nome_gerente* e no salário médio de todos os funcionários que trabalham para esse gerente. Explique por que o sistema de banco de dados não permite que atualizações sejam expressas em termos dessa view.

3.15 Escreva uma consulta SQL, sem usar uma cláusula **with**, para encontrar todas as agências em que o depósito em conta total seja menor do que a média dos depósitos em conta totais de todas as agências.

a. Usando uma consulta aninhada na cláusula **from**.
b. Usando uma consulta aninhada na cláusula **having**.

3.16 Cite duas razões por que valores nulos precisam ser introduzidos no banco de dados.

3.17 Mostre como expressar a operação **coalesce** do Exercício 3.4 usando a operação **case**.

3.18 Dê uma definição de esquema SQL para o banco de dados de funcionários da Figura 3.12. Escolha um domínio apropriado para cada atributo e uma chave primária apropriada para cada esquema de relação.

3.19 Usando as relações do nosso banco de dados de banco, escreva expressões SQL para definir as seguintes views:

a. Uma view contendo os números de conta e os nomes de cliente (mas não os saldos) para todas as contas na agência Deer Park.
b. Uma view contendo os nomes e endereços de todos os clientes que têm uma conta com o banco, mas que não têm um empréstimo.
c. Uma view contendo o nome e o saldo médio de cada cliente da agência Rock Ridge.

3.20 Para cada uma das views que você definiu no Exercício 3.19, explique como as atualizações seriam realizadas (se elas devessem ser permitidas).

3.21 Considere o seguinte esquema relacional.

funcionário (*num_emp*, *nome*, *escritório*, *idade*)
livros (*isbn*, *título*, *autores*, *editora*)
empréstimo (*num_emp*, *isbn*, *data*)

Escreva as seguintes consultas em SQL.

a. Imprima os nomes dos funcionários que tomaram emprestado qualquer livro publicado pela McGraw-Hill.
b. Imprima os nomes dos funcionários que tomaram emprestados todos os livros publicados pela McGraw-Hill.
c. Para cada editora, imprima os nomes dos funcionários que tomaram emprestados mais de cinco livros dessa editora.

3.22 Considere o seguinte esquema relacional.

aluno (*id_aluno*, *nome_aluno*)
registrado (*id_aluno*, *id_curso*)

Escreva uma consulta SQL para listar o ID e o nome de cada aluno, juntamente com o número total de cursos para os quais ele está registrado. Os alunos que não estão registrados para curso algum também precisam ser listados, com o número de cursos registrados mostrado como 0.

3.23 Suponha que tenhamos uma relação *marcas* (*id_aluno*, *pontuação*). Escreva uma consulta SQL para encontrar a *avaliação bruta* de cada aluno. Ou seja, todos os alunos com a marca mais alta obtêm uma avaliação de 1, os alunos com a próxima marca mais alta obtêm uma avaliação de 2 e assim por diante. Dica: divida a tarefa em partes, usando a cláusula with.

Notas bibliográficas

A versão original da SQL, chamada Sequel 2, é descrita por Chamberlin e outros [1976]. A Sequel 2 foi derivada das linguagens Square (Boyce e outros [1975] e Chamberlin e Boyce [1974]). O American National Standard SQL-86 é descrito na ANSI [1986]. A definição IBM Systems Application Architecture da SQL é definida por IBM [1987]. Os padrões oficiais para a SQL-89 e SQL-92 estão disponíveis como ANSI [1989] e ANSI [1992], respectivamente.

Livros-texto com descrições da linguagem SQL-92 incluem Date e Darwen [1997], Melton e Simon [1993] e Cannan e Otten [1993]. Date e Darwen [1997] e Date [1993a] incluem uma crítica à SQL-92.

Os livros sobre a SQL:1999 incluem Melton e Simon [2001] e Melton [2002]. Fisenberg e Melton [1999] fornecem uma sinopse da SQL:1999. Donahoo e Speegle [2005] abordam a SQL de uma perspectiva dos desenvolvedores. Eisenberg *et al.* [2004] fornecem uma sinopse da SQL:2003.

Os padrões SQL:1999 e SQL:2003 são publicados como uma coleção de documentos dos padrões ISO/IEC, que são descritos em mais detalhes no Capítulo 23. Os documentos de padrões são densamente preenchidos com informações e são difíceis de ler, destinando-se principalmente a implementadores de sistemas de banco de dados. Os documentos de padrões estão disponíveis para compra eletronicamente no site `http://webstore.ansi.org`.

Muitos produtos de banco de dados aceitam recursos SQL além dos especificados no padrão e não aceitam alguns recursos do padrão. Mais informações sobre esses recursos podem ser encontradas nos manuais do usuário SQL dos respectivos produtos.

O processamento das consultas SQL, incluindo algoritmos e questões de desempenho, é discutido nos Capítulos 13 e 14. As referências bibliográficas sobre esses assuntos aparecem nesses capítulos.

As regras usadas pela SQL para determinar a capacidade de atualização de uma view, e como as atualizações são refletidas nas relações de banco de dados subjacentes, são definidas pelo padrão SQL:1999 e são resumidas em Melton e Simon [2001].

CAPÍTULO 4

SQL avançada

No Capítulo 3, fornecemos uma abordagem detalhada da estrutura básica da SQL. A linguagem SQL cresceu, desde o final da década de 1970, de uma simples linguagem com alguns recursos para uma linguagem bastante complexa, com recursos para satisfazer muitos tipos de usuários diferentes. Neste capítulo, veremos alguns dos recursos avançados da SQL. Para nossa conveniência, continuaremos a usar em nossos exemplos o esquema de banco, reproduzido na Figura 4.1.

Tipos de dados e esquemas da SQL

Vimos que um tipo, isto é, um domínio de possíveis valores, precisa estar associado a cada atributo. No Capítulo 3, examinamos vários tipos de dados internos aceitos na SQL, como os tipos integer, real e character. Existem outros tipos de dados internos aceitos pela SQL, que descreveremos a seguir. Também veremos como criar tipos básicos definidos pelo usuário na SQL.

Tipos de dados internos na SQL

Além dos tipos de dados básicos que apresentamos na seção "Definição de dados" do Capítulo 3, o padrão SQL aceita outros tipos de dados internos, incluindo:

- **date**: uma data de calendário contendo um ano (de quatro dígitos), mês e dia do mês.
- **time**: a hora do dia, em horas, minutos e segundos. Uma variante, **time**(*p*), pode ser usada para especificar o número de dígitos fracionários para os segundos (o padrão é 0). Também é possível armazenar informações de fuso horário juntamente com a hora especificando **time whit timezone**.
- **timestamp**: uma combinação de **date** e **time**. Uma variante, **timestamp**(*p*), pode ser usada para especificar o número de dígitos fracionários para os segundos (o padrão é 6). As informações de fuso horário também são armazenadas se **with timezone** for especificado.

Os valores de data e hora podem ser especificados como:

date '2001-04-25'
time '09:30:00'
timestamp '2001-04-25 10:29:01.45'

As datas precisam ser especificadas no formato aaaa-mm-dd, ou seja, o ano, seguido do mês, seguido do dia. O campo dos segundos de **time** ou **timestamp** pode ter uma parte fracionária, como nessa estampa de tempo.

agência (<u>*nome_agência*</u>, *cidade_agência*, *ativo*)
cliente (<u>*nome_cliente*</u>, *rua_cliente*, *cidade_cliente*)
empréstimo (<u>*número_empréstimo*</u>, *nome_agência*, *quantia*)
tomador (<u>*nome_cliente*</u>, <u>*número_empréstimo*</u>)
conta (<u>*número_conta*</u>, *nome_agência*, *saldo*)
depositante (<u>*nome_cliente*</u>, <u>*número_conta*</u>)

Figura 4.1 Esquema da instituição bancária.

Podemos usar uma expressão da forma **cast** *e* as *t* para converter uma string de caracteres (ou expressão com valor de string) *e* no tipo *t*, onde *t* é **date**, **time** ou **timestamp**. A string precisa estar no formato apropriado como ilustrado anteriormente. Quando necessário, as informações de fuso horário são obtidas das configurações do sistema.

Para extrair campos individuais de um valor de **date** ou **time** *d*, podemos usar **extract** (*campo* **from** *d*), onde *campo* pode ser **ano**, **mês**, **dia**, **hora**, **minuto** ou **segundo**. As informações de fuso horário podem ser extraídas usando **timezone_hour** e **timezone_minute**.

A SQL também define várias funções úteis para obter a data e hora atual. Por exemplo, **current_date** retorna a data atual, **current_time** retorna a hora atual (com fuso horário) e **localtime** retorna a hora local atual (sem fuso horário). As estampas de tempo (data mais hora) são retornadas por **current_timestamp** (com fuso horário) e **localtimestamp** (data e hora locais sem fuso horário).

A SQL permite operações de comparação em todos os tipos relacionados aqui e permite operações aritméticas e de comparação nos vários tipos numéricos. A SQL também fornece um tipo de dados chamado **interval** e permite cálculos baseados em datas e horas e em intervalos. Por exemplo, se *x* e *y* são do tipo **date**, então, *x* – *y* é um intervalo cujo valor é o número de dias da data *x* até a data *y*. Da mesma forma, somar ou subtrair um intervalo de uma data ou hora resulta em uma data ou hora, respectivamente.

Muitas vezes, é útil comparar valores de diferentes tipos **compatíveis**. Como uma ilustração, suponha que o tipo de *nome_cliente* seja uma string de caractere de tamanho 20, e o tipo de *nome_agência* seja uma string de caractere de tamanho 15. Embora os tamanhos de string possam diferir, o padrão SQL irá considerar os dois tipos compatíveis. Como outro exemplo, já que todo inteiro pequeno também é um inteiro, faz sentido uma comparação $x < y$, onde x é um inteiro pequeno e y é um inteiro (ou vice-versa). Podemos fazer essa comparação convertendo o inteiro pequeno x em um inteiro. Uma transformação desse tipo é chamada de **coerção de tipo**. A coerção de tipo é usada rotineiramente nas linguagens de programação comuns, bem como nos sistemas de banco de dados.

Tipos definidos pelo usuário

A SQL aceita duas formas de tipos de dados definidos pelo usuário. A primeira forma, que abordaremos aqui, é chamada de **tipos distintos**. A outra forma, chamada **tipos de dados estruturados**, permite a criação de tipos de dados complexos com estruturas de registro, arrays e multiconjuntos aninhados. Não estudaremos os tipos de dados estruturados neste capítulo, mas os descreveremos posteriormente, no Capítulo 9.

É possível que vários atributos tenham o mesmo tipo de dados. Por exemplo, os atributos *nome_cliente* e *nome_funcionário* podem ter o mesmo domínio: o conjunto de todos os nomes de pessoa. Entretanto, os domínios de *saldo* e *nome_agência* certamente podem ser diferentes. Talvez seja menos claro se *nome_cliente* e *nome_agência* devem ter o mesmo domínio. No nível de implementação, tanto os nomes de cliente quanto os nomes de agência são strings de caractere. Contudo, normalmente não consideraríamos a consulta "Encontre todos os clientes que têm o mesmo nome de uma agência" como sendo uma consulta significativa. Portanto, se virmos o banco de dados no nível conceitual em vez de no nível físico, *nome_cliente* e *nome_agência* devem ter domínios distintos.

Mais importante, em um nível prático, atribuir o nome de um cliente a uma agência provavelmente é um erro de programação; da mesma forma, comparar um valor monetário expresso em dólares diretamente com um valor monetário expresso em libras também é certamente um erro de programação. Um bom sistema de tipo deve ser capaz de detectar essas atribuições ou comparações. Para aceitar essas verificações, a SQL fornece a noção de **tipos distintos**.

A cláusula **create type** pode ser usada para definir novos tipos. Por exemplo, as instruções:

```
create type Dólares as numeric(12,2) final
create type Libras as numeric(12,2) final
```

definem os tipos definidos pelo usuário *Dólares* e *Libras* para serem números decimais com um total de 12 dígitos, dois dos quais são colocados após a vírgula decimal. (A palavra-chave **final** não é realmente significativa nesse contexto, mas é necessária pelo padrão SQL:1999 por questões que não devemos explicar agora; algumas implementações permitem que a palavra-chave **final** seja omitida.) Os tipos recém-criados podem, então, ser usados como tipos de atributos de relações. Por exemplo, poderíamos declarar a tabela *conta* como:

```
create table conta
    (número_conta char(10),
    nome_agência char(15),
    saldo Dólares)
```

Uma tentativa de atribuir um valor do tipo *Dólares* a uma variável do tipo *Libra* resultaria em um erro de tempo de compilação, embora ambos sejam do mesmo tipo numérico. Essa atribuição provavelmente é devido a um erro de programação, em que o programador se esqueceu das diferenças de moeda. Declarar tipos diferentes para moedas diferentes ajuda a detectar esses tipos de erro.

Como um resultado da verificação de tipo forte, a expressão (*conta.saldo*+20) não seria aceita, já que o atributo e a constante de inteiro 20 possuem tipos diferentes. Os valores de um tipo podem ser convertidos em outro domínio, como ilustrado a seguir.

cast (*conta_saldo* **to** *numeric*(12,2))

Poderíamos fazer a adição no tipo numeric, mas, para salvar o resultado em um atributo do tipo *Dólares*, precisaríamos usar outra expressão cast de modo a converter o tipo novamente em *Dolares*.

A SQL também fornece cláusulas **drop type** e **alter type** para descartar ou modificar tipos que tenham sido criados anteriormente.

Mesmo antes de os tipos definidos pelo usuário serem acrescentados à linguagem (na SQL:1999), a SQL tinha uma noção semelhante mas ligeiramente distinta de **tipo de domínio** (introduzida na SQL-92). Poderíamos definir um tipo de domínio *DDólares* desta forma:

create domain *DDólares* **as numeric**(12,2)

O tipo de domínio *DDólares* pode ser usado como tipo de atributo, exatamente como usamos o tipo *Dólares*. Entretanto, existem duas diferenças significantes entre tipos e domínios:

1. Os domínios podem ter restrições, como **not null**, especificadas neles, e podem ter valores-padrão definidos para variáveis do tipo domínio, enquanto os tipos definidos pelo usuário não podem ter restrições ou valores-padrão especificados neles. Os tipos definidos pelo usuário são projetados para serem usados não apenas para especificar tipos de atributo, mas também nas extensões procedurais da SQL, em que pode não ser possível impor restrições. Retornarmos à questão das restrições em domínios mais tarde, na seção "A cláusula check".
2. Os domínios não são fortemente tipificados. Como resultado, os valores de um tipo de domínio podem ser atribuídos aos valores de outro tipo de domínio, desde que os tipos básicos sejam compatíveis.

Tipos de objeto grande

Muitas aplicações de banco de dados de última geração precisam armazenar atributos que podem ser grandes (da ordem de muitos kilobytes!), como a fotografia de uma pessoa, ou muito grandes (da ordem de muitos megabytes ou mesmo gigabytes), como uma imagem médica ou videoclipe de alta resolução. Portanto, a SQL fornece novos tipos de dados de objeto grande para dados de caractere (**clob**) e dados binários (**blob**). As letras "lob" nesses tipos de dados significam "Large OBject". Por exemplo, podemos declarar os atributos

revisão_livro **clob**(10KB)
image **blob**(10MB)
filme **blob**(2GB)

Executar uma consulta SQL normalmente recuperaria uma ou mais linhas do resultado para a memória. Os objetos grandes geralmente são usados em aplicações externas, e, para objetos muito grandes (vários megabytes ou gigabytes), é ineficiente ou impraticável recuperar um objeto grande inteiro para a memória. Em vez disso, uma aplicação normalmente usaria uma consulta SQL a fim de recuperar um "localizador" para um objeto grande e, depois, usaria o localizador para manipular o objeto por meio da linguagem host. Por exemplo, a interface de programa de aplicação JDBC (descrita na seção "JDBC") permite que um localizador seja buscado em vez de um objeto grande inteiro; o localizador, então, pode ser usado para buscar o objeto grande em pequenas partes, em vez de todo de uma vez, muito semelhante a ler dados de um arquivo de sistema operacional usando uma chamada de função `read`.

Esquemas, catálogos e ambientes

Para entender a motivação para esquemas e catálogos, considere como os arquivos são nomeados em um sistema de arquivos. Os primeiros sistemas de arquivos eram planos; ou seja, todos os arquivos eram armazenados em um único diretório. Os sistemas de arquivos atuais, é claro, possuem uma estrutura de diretórios, com arquivos armazenados dentro de subdiretórios. Para nomear um arquivo unicamente, precisamos especificar o nome de caminho completo do arquivo, por exemplo, `/usuários/avi/livro-bancodados/cap04.doc`.

Como os antigos sistemas de arquivos, os primeiros sistemas de banco de dados também tinham um único espaço de nomes para todas as relações. Os usuários precisavam se coordenar para assegurar que não tentaram usar o mesmo nome para diferentes relações. Os sistemas de banco de dados modernos fornecem uma hierarquia de três níveis para nomear relações. O nível superior da hierarquia consiste nos **catálogos**, cada um podendo conter **esquemas**. Os objetos SQL, como relações e views, estão contidos em um **esquema**. (Algumas implementações de banco de dados usam o termo "banco de dados" no lugar do termo "catálogo".)

Para realizar quaisquer ações em um banco de dados, um usuário (ou um programa) precisa primeiro se *conectar* ao banco de dados. O usuário precisa fornecer o nome de

usuário e normalmente uma senha secreta para verificar a identidade do usuário. Cada usuário tem um catálogo e um esquema padrão, e a combinação é única ao usuário. Quando um usuário se conecta a um sistema de banco de dados, o catálogo e o esquema padrão são configurados para a conexão; isso corresponde ao diretório atual ser definido como o diretório home do usuário quando este se registra em um sistema operacional.

Para identificar uma relação unicamente, um nome de três partes precisa ser usado, por exemplo,

```
catálogo5.esquema_banco.conta
```

Podemos omitir o componente catálogo, caso em que a parte do catálogo do nome é considerada como sendo o catálogo padrão para a conexão. Assim, se `catálogo5` é o catálogo padrão, podemos usar apenas `esquema_banco.conta` para identificar a mesma relação unicamente. Além disso, também podemos omitir o nome do esquema, e a parte do esquema do nome é novamente considerada como sendo o esquema padrão para a conexão. Portanto, podemos usar apenas `conta` se o catálogo padrão for `catálogo5` e o esquema padrão for `esquema_banco`.

Com diversos catálogos e esquemas disponíveis, diferentes aplicações e diferentes usuários podem trabalhar independentemente sem se preocupar com conflitos de nome. Além disso, várias versões de uma aplicação – uma versão de produção, outra versão de teste – podem ser executadas no mesmo sistema de banco de dados.

O catálogo e o esquema padrão são parte de um ambiente SQL que é configurado para cada conexão. O ambiente também contém o identificador de usuário (também chamado de *identificador de autorização*). Todas as instruções SQL comuns, incluindo as instruções DDL e DML, operam no contexto de um esquema. Podemos criar e descartar esquemas com as instruções **create schema** e **drop schema**, respectivamente. A criação e o descarte de catálogos dependem da implementação e não são parte do padrão SQL.

Restrições de integridade

As restrições de integridade garantem que as mudanças feitas no banco de dados por usuários autorizados não resultem em uma perda da consistência dos dados. Portanto, as restrições de integridade protegem contra danos acidentais no banco de dados.

Exemplos de restrições de integridade são:

- Um saldo de conta não pode ser nulo.
- Nenhum par de contas pode ter o mesmo número de conta.
- Cada número de conta na relação *depositante* precisa ter um número de conta correspondente na relação *conta*.
- O salário por hora de um funcionário de banco precisa ser, pelo menos, $6,00.

Em geral, uma restrição de integridade pode ser um predicado arbitrário pertencente ao banco de dados. Entretanto, como os predicados arbitrários podem ser difíceis de testar, a maioria dos sistemas de banco de dados permite que se especifiquem restrições de integridade que podem ser testadas com um mínimo de overhead. Estudaremos algumas formas de restrições de integridade nesta seção. No Capítulo 7 estudaremos outra forma de restrição de integridade, chamada **dependências funcionais**, que é usada principalmente no processo do projeto de esquema.

Restrições em uma única relação

Na seção "Definição de dados" do Capítulo 3 descrevemos como definir tabelas usando o comando **create table**. O comando **create table** também pode incluir instruções de restrição de integridade. Além da restrição de "chave primária", existem várias outras restrições que podem ser incluídas no comando **create table**. As restrições de integridade permitidas incluem

- **not null**
- **unique**
- **check**(<predicado>)

Discutiremos cada um desses tipos de restrições nas seções a seguir.

Restrição not null

Como vimos no Capítulo 2, o valor *nulo* é um membro de todos os domínios e, conseqüentemente, por padrão, é um valor válido para todos os atributos na SQL. Para certos atributos, no entanto, os valores nulos podem ser inapropriados. Considere uma tupla na relação *conta* onde *número_conta* seja nulo. Essa tupla fornece informações de conta para uma conta desconhecida; portanto, ela não contém informações úteis. De igual maneira, não desejaríamos que o saldo da conta fosse nulo. Em casos como esse, desejaríamos proibir valores nulos, o que poderíamos fazer restringindo o domínio dos atributos *número_conta* e *saldo* para excluir valores nulos, declarando-os do seguinte modo:

número_conta **char**(10) **not null**
saldo **numeric**(12,2) **not null**

A especificação **not null** proíbe a inserção de um valor nulo para esse atributo. Qualquer modificação de

banco de dados que causaria a inserção de um nulo em um atributo declarado para ser **not null** gera um diagnóstico de erro.

Existem muitas situações em que queremos evitar valores nulos. Em especial, a SQL proíbe valores nulos na chave primária de um esquema de relação. Portanto, em nosso exemplo de banco, na relação *conta*, se o atributo *número_conta* for declarado como a chave primária para *conta*, ele não poderá tomar um valor nulo. Como resultado, ele não precisaria ser explicitamente declarado para ser **not null**.

A especificação **not null** também pode ser aplicada a uma declaração de domínio definida pelo usuário; consequentemente, os atributos desse tipo de domínio não poderiam tomar um valor nulo. Portanto, se quisessemos que nosso domínio *Dólares* não tomasse valores nulos, poderíamos declará-lo da seguinte maneira:

```
create domain Dólares numeric(12,2) not null
```

Restrição unique

A SQL também aceita uma restrição de integridade

unique $(A_{j1}, A_{j2}, \ldots, A_{jm})$

A especificação **unique** diz que os atributos $A_{j1}, A_{j2}, \ldots, A_{jm}$ formam uma chave candidata; ou seja, nenhum par de tuplas na relação pode ser igual em todos os atributos de chave primária. Entretanto, os atributos de chave candidata podem ser nulos, a menos que tenham sido explicitamente declarados como sendo **not null**. Lembre-se de que um valor nulo não se iguala a qualquer outro valor. (O tratamento dos nulos aqui é o mesmo da construção **unique** definida na seção "Teste da ausência de tuplas duplicatas" do Capítulo 3.)

A cláusula check

A cláusula **check** na SQL pode ser aplicada a declarações de relação, bem como a declarações de domínio. Quando aplicada a uma declaração de relação, a cláusula **check**(*P*) especifica um predicado *P* que precisa ser satisfeito por tupla em uma relação.

Um uso comum da cláusula **check** é assegurar que os valores de atributo satisfaçam condições especificadas, efetivamente criando um poderoso sistema de tipos. Por exemplo, uma cláusula **check**(*ativo*>=0) no comando **create table** para a relação *agência* garantiria que o valor de *ativo* seja não negativo.

Como outro exemplo, considere o seguinte:

```
create table aluno
        (nome          char(15) not null,
        id_aluno       char(10),
        nível_grau     char(15),
        primary key (id_aluno),
        check (nível_grau in ('Bacharelado',
'Mestrado', 'Doutorado')))
```

Aqui, usamos a cláusula **check** para simular um tipo enumerado, especificando que *nível_grau* precisa ser 'Bacharelado', 'Mestrado' ou 'Doutorado'.

Quando aplicada a um domínio, a cláusula **check** permite que o projetista do esquema especifique um predicado que precisa ser satisfeito por qualquer valor atribuído a uma variável cujo tipo é o domínio.

Por exemplo, uma cláusula **check** pode garantir que um domínio de salário permita apenas valores maiores do que um valor especificado (como o salário mínimo):

```
create domain Salário numeric(5,2)
        constraint teste_valor_salário check(value >= 6,00)
```

O domínio *Salário* possui uma restrição que garante que o salário seja maior ou igual a 6,00. A cláusula **constraint** *teste_valor_salário* é opcional e é usada para atribuir o nome *teste_valor_salário* à restrição. O nome é usado pelo sistema para indicar a restrição que uma atualização violou.

Como outro exemplo, um domínio pode ser restrito para conter apenas um conjunto especificado de valores usando a cláusula **in**:

```
create domain TipoConta char(10)
        constraint teste_tipo_conta
                check(value in ('Corrente', 'Poupança'))
```

Desse modo, a cláusula **check** permite que atributos e domínios sejam restritos de maneiras poderosas que a maioria dos sistemas de tipificação de linguagem de programação não permite.

As condições **check** anteriores podem ser testadas com bastante facilidade, quando uma tupla é inserida ou modificada. Entretanto, geralmente as condições **check** podem ser mais complexas (e mais difíceis de testar), já que as subconsultas que fazem referência a outras relações são permitidas na condição **check**. Por exemplo, essa restrição poderia ser especificada na relação *depósito*:

```
check (nome_agência in (select nome_agência from agência))
```

A condição **check** verifica se o *nome_agência* em cada tupla na relação *depósito* é realmente o nome de uma agência na relação *agência*. Portanto, a condição precisa ser verifi-

cada não só quando uma tupla é inserida ou modificada em *depósito*, mas também quando a relação *agência* muda (nesse caso, quando uma tupla é excluída ou modificada na relação *agência*).

A restrição anterior é, na verdade, um exemplo de uma classe de restrições chamadas de *integridade referencial*. Na próxima seção veremos essas restrições, bem como uma maneira mais simples de especificá-las na SQL.

As condições check complexas podem ser úteis quando queremos assegurar a integridade dos dados, mas devemos usá-las com cuidado, já que podem ser difíceis de testar.

Integridade referencial

Muitas vezes, queremos garantir que um valor que aparece em uma relação para um determinado conjunto de atributos também apareça para um certo conjunto de atributos em outra relação. Essa condição é chamada de integridade referencial.

Chaves estrangeiras podem ser especificadas como parte da instrução SQL **create table** usando a cláusula foreign key. Ilustramos as declarações de chave estrangeira usando a definição DDL SQL de parte do nosso banco de dados de banco, mostrado na Figura 4.2. A definição da tabela *conta* possui uma declaração "**foreign key** (*nome_agência*) **references** *agência*". Essa declaração de chave estrangeira especifica que, para cada tupla na relação *conta*, o nome de agência especificado na tupla precisa existir na relação *agência*. Sem essa restrição, é possível que uma conta especifique um nome de conta inexistente.

Mais geralmente, consideremos que $r_1(R_1)$ e $r_2(R_2)$ sejam relações com chaves primárias K_1 e K_2, respectivamente (lembre-se de que R_1 e R_2 indicam o conjunto dos atributos de r_1 e r_2, respectivamente). Dizemos que um subconsulta α de R_2 é uma **chave estrangeira** referenciando K_1 na relação r_1 se for exigido que, para cada tupla t_2 em r_2, precise existir uma tupla t_1 em r_1 tal que $t_1[K_1] = t_2[\alpha]$. Exigências desse tipo são chamadas de **restrições de integridade referencial** ou **dependências de subconsulta**. O último termo surge porque a restrição de integridade referencial anterior pode ser escrita como $\Pi_\alpha (r_2) \subseteq \Pi_{K_1} (r_1)$. Note que, para uma restrição de integridade referencial fazer sentido,

```
create table cliente
    (nome_cliente    char(20),
     rua_cliente     char(30),
     cidade_cliente  char(30),
     primary key (nome_cliente))

create table agência
    (nome_agência    char(15),
     cidade_agência  char(30),
     ativo           numeric(16,2),
     primary key (nome_agência),
     check (ativo >= 0))

create table conta
    (número_conta    char(10),
     nome_agência    char(15),
     saldo           numeric(12,2),
     primary key (número_conta),
     foreign key (nome_agência) references agência,
     check (saldo >= 0))

create table depositante
    (nome_cliente    char(20),
     número_conta    char(10),
     primary key (nome_cliente, número_conta),
     foreign key (nome_cliente) references cliente,
     foreign key (número_conta) references conta)
```

Figura 4.2 Definição de dados SQL para parte do banco de dados de banco.

α e K_1 precisam ser conjuntos de atributos compatíveis; isto é, ou α precisa ser igual a K_1 ou eles precisam conter o mesmo número de atributos, e os tipos de atributos correspondentes precisam ser compatíveis (aqui consideramos que α e K_1 estejam ordenados).

Como padrão, na SQL, uma chave estrangeira referencia os atributos de chave primária da tabela referenciada. A SQL também aceita uma versão da cláusula **references** em que uma lista de atributos da relação referenciada pode ser especificada explicitamente. A lista de atributos especificada, no entanto, precisa ser declarada como uma chave candidata da relação referenciada.

É possível usar a seguinte forma curta como parte de uma definição de atributo para declarar que o atributo forma uma chave estrangeira:

nome_agência **char**(15) **references** *agência*

Quando uma restrição de integridade referencial é violada, o procedimento normal é rejeitar a ação que causou a violação (ou seja, a transação realizando a ação update é revertida). Entretanto, uma cláusula **foreign key** pode especificar que, se uma ação delete ou update na relação referenciada violar a restrição, então, em vez de rejeitar a ação, o sistema precisa tomar providências para mudar a tupla na relação referenciadora a fim de restaurar a restrição. Considere esta definição de uma restrição de integridade na relação *conta*:

```
create table conta
  (...
  foreign key (nome_agência) references agência
                             on delete cascade
                             on update cascade,
  ...)
```

Devido à cláusula **on delete cascade** associada à declaração de chave estrangeira, se a exclusão de uma tupla em *agência* fizer com que essa restrição de integridade referencial seja violada, o sistema não rejeitará a exclusão. Em vez disso, a exclusão "transborda em cascata" para a relação *conta*, excluindo a tupla que faz referência à agência que foi excluída. Da mesma maneira, o sistema não rejeita uma atualização em um campo referenciado pela restrição se ela violar a restrição; em vez disso, o sistema também atualiza o campo *nome_agência* nas tuplas referenciadoras em *conta* para o novo valor. A SQL também permite que a cláusula **foreign key** especifique ações diferentes de **cascade** se a restrição for violada: o campo referenciador (aqui, *nome_agência*) pode ser definido em nulo (usando **set null** em vez de **cascade**), ou no valor-padrão para o domínio (usando **set default**).

Se houver uma cadeia de dependências de chave estrangeira por meio de várias relações, uma exclusão ou atualização em uma ponta da cadeia pode se propagar por toda a cadeia. Um caso interessante, em que a restrição **foreign key** na relação referencia a mesma relação, aparece no Exercício prático 4.4. Se uma atualização ou exclusão em cascata causar uma violação de restrição que não pode ser tratada por uma operação em cascata subseqüente, o sistema aborta a transação. Como resultado, todas as mudanças causadas pela transação e suas ações em cascata são desfeitas.

Valores **null** complicam a semântica das restrições de integridade referencial na SQL. Os atributos para chaves estrangeiras podem ser nulos, desde que não tenham sido declarados como não nulos. Se todas as colunas de uma chave estrangeira forem não nulas em uma determinada tupla, a definição comum das restrições de chave estrangeira é usada para essa tupla. Se qualquer uma das colunas de chave estrangeira for nula, a tupla é definida automaticamente para satisfazer a restrição.

Como essa definição nem sempre será a escolha certa, a SQL também fornece construções que permitem mudar o comportamento com valores nulos; não discutimos as construções aqui.

As restrições de integridade podem ser acrescentadas a uma relação existente usando o comando **alter table** *nome_tabela* **add** *restrição*, onde *restrição* pode ser qualquer uma das restrições que vimos. Quando esse comando é executado, o sistema primeiramente se certifica de que a relação satisfaz a restrição especificada. Em caso positivo, a restrição é acrescentada à relação; caso contrário, o comando é rejeitado.

As transações podem consistir em várias etapas, e as restrições de integridade podem ser violadas temporariamente após uma etapa, mas uma etapa seguinte pode remover a violação. Por exemplo, suponha que tenhamos uma relação *pessoa*, com chave primária *nome* e um atributo *cônjuge*, e suponha que *cônjuge* seja uma chave estrangeira em *pessoa*. Ou seja, a restrição diz que o atributo *cônjuge* precisa conter um nome que esteja presente na tabela *pessoa*. Suponha que desejamos notar o fato de que John e Mary sejam casados um com o outro inserindo duas tuplas, uma para John e uma para Mary, nessa relação. A inserção da primeira tupla violaria a restrição de chave estrangeira, independentemente de qual das duas tuplas seja inserida primeiro. Após a segunda tupla ser inserida, a restrição de chave estrangeira se aplicaria novamente.

Para tratar dessas situações, o padrão SQL permite que uma cláusula **initially deferred** seja incluída em uma especificação de restrição; a restrição, então, seria verificada no final da transação e não nas etapas intermediárias.[1] Alternati-

[1] Podemos contornar o problema nesse exemplo de outra maneira, se o atributo *cônjuge* puder ser definido como nulo: definimos os atributos de cônjuge como nulos ao inserir as tuplas para John e Mary e os atualizamos mais tarde. Todavia, essa técnica é um tanto confusa e não funciona se os atributos não puderem ser definidos como nulos.

vamente, uma restrição pode ser especificada como **deferrable**, o que significa que, por padrão, ela é verificada imediatamente, mas pode ser adiada quando desejado. Para restrições declaradas como deferrable (adiáveis), executar uma instrução **set constraints** *lista-restrição* **deferred** como parte de uma transação faz com que a verificação das restrições especificadas seja adiada para o final da transação.

Entretanto, você deve estar ciente de que o comportamento padrão é verificar restrições imediatamente, e muitas implementações de banco de dados não aceitam verificação de restrição adiada.

Afirmações

Uma **afirmação** é um predicado expressando uma condição que desejamos que o banco de dados sempre satisfaça. As restrições de domínio e as restrições de integridade referencial são formas especiais de afirmações. Dedicamos uma atenção especial a essas formas de afirmação porque elas são facilmente testadas e se aplicam a uma ampla faixa de aplicações de banco de dados. Contudo, existem muitas restrições que não podemos expressar usando apenas essas formas especiais. Dois exemplos dessas restrições são:

- A soma de todas as quantias de empréstimo para cada agência precisa ser menor que a soma de todos os saldos de conta na agência.
- Cada empréstimo tem pelo menos um cliente que mantém uma conta com um saldo mínimo de $1.000.

Uma afirmação na SQL tem a forma

create assertion <nome-afirmação> **check** <predicado>

Na Figura 4.3, mostramos como os dois exemplos de restrições podem ser escritos na SQL. Como a SQL não fornece uma construção "para todo X, $P(X)$", onde P é um predicado, somos obrigados a implementar a restrição por uma construção equivalente, "não existe X tal que não $P(X)$", que pode ser expressa na SQL.

Quando uma afirmação é criada, o sistema testa sua validade. Se a afirmação for válida, então, qualquer modificação futura no banco de dados só será permitida se ela não fizer com que a afirmação seja violada. Esse teste pode introduzir uma quantidade significativa de overhead se afirmações complexas tiverem sido feitas. Conseqüentemente, as afirmações devem ser usadas com muito critério. O alto overhead de testar e manter afirmações levou alguns desenvolvedores de sistema a omitir o suporte para afirmações gerais, ou a fornecer formas especializadas de afirmação que sejam mais fáceis de testar.

Autorização

Podemos atribuir a um usuário várias formas de autorizações sobre partes do banco de dados. Por exemplo,

- Autorização para ler dados
- Autorização para inserir novos dados
- Autorização para atualizar dados
- Autorização para excluir dados

Cada um desses tipos de autorização é chamado de um **privilégio**. Podemos autorizar um usuário a todos, a nenhum ou a uma combinação desses privilégios sobre partes específicas de um banco de dados, como uma relação ou uma view.

```
create assertion restrição_soma check
  (not exists (select * from agência
    where (select sum(quantia) from empréstimo
        where empréstimo.nome_agência = agência.nome_agência)
      >= (select sum(saldo) from conta
        where conta.nome_agência = agência.nome_agência)))

create assertion restrição_saldo check
  (not exists (select * from empréstimo
    where not exists (select *
      from tomador, depositante, conta
      where empréstimo.número_empréstimo = tomador.número_empréstimo)
        and tomador.nome_cliente = depositante.nome_cliente
        and depositante.número_conta = conta.número_conta
        and conta.saldo >= 1000)))
```

Figura 4.3 Dois exemplos de afirmação.

O padrão SQL inclui os privilégios **select**, **insert**, **update** e **delete**. O privilégio **select** autoriza o usuário a ler dados. Além dessas formas de privilegios para acessar dados, a SQL aceita vários outros privilégios, como o de criar, excluir ou modificar relações, e o de executar procedimentos. Discutiremos esses privilégios posteriormente, na seção "Autorização em SQL" do Capítulo 8. O privilégio **all privileges** pode ser usado como uma forma resumida para todos aqueles que são permitidos. Um usuário que cria uma nova relação recebe automaticamente todos os privilégios sobre essa relação.

Um usuário que possui alguma forma de autorização pode concedê-la a outros usuários, ou revogar uma autorização que foi concedida anteriormente.

A linguagem de definição de dados da SQL inclui comandos para conceder e revogar privilégios. A instrução **grant** é usada para conferir autorização. A forma básica dessa instrução é:

grant <lista-privilégios> **on** <nome-relação ou nome-view> **to** <lista_usuários/papéis>

A *lista de privilégios* permite a concessão de vários privilégios em um único comando. O conceito de papéis é discutido mais tarde, na seção "Autorização em SQL" do Capítulo 8.

A instrução **grant** a seguir concede aos usuários de banco de dados John e Mary o privilégio **select** sobre a relação *conta*:

grant select on *conta* **to** *John, Mary*

O privilégio **update** pode ser concedido sobre todos os atributos da relação ou apenas a alguns. Se o privilegio **update** estiver incluído em uma instrução **grant**, a lista dos atributos sobre os quais o privilégio deve ser concedido aparece opcionalmente entre parênteses, imediatamente após a palavra-chave **update**. Se a lista de atributos for omitida, o privilégio **update** será concedido a todos os atributos da relação.

A seguinte instrução **grant** confere aos usuários John e Mary autorização de atualização sobre o atributo *quantia* da relação *empréstimo*:

grant update (*quantia*) **on** *empréstimo* **to** *John, Mary*

O privilégio **insert** também pode especificar uma lista de atributos; quaisquer inserções na relação precisam especificar apenas esses atributos, e o sistema fornece valores-padrão a cada um dos atributos restantes (se um padrão estiver definido para o atributo) ou os define como nulos.

O nome de usuário **public** se refere a todos os usuários atuais e futuros do sistema. Assim, privilégios concedidos a **public** são implicitamente concedidos a todos os usuários atuais e futuros.

Como padrão, um usuário/papel que recebe um privilégio não está autorizado a conceder esse privilégio a outro usuário/papel. A SQL permite uma concessão de privilégio para especificar que o receptor pode conceder o privilégio a outro usuário. Descreveremos esse recurso em mais detalhes na seção "Autorização em SQL" do Capítulo 8.

Para revogar uma autorização, usamos a instrução **revoke**. Ela tem uma forma quase idêntica à da instrução **grant**:

revoke <lista-privilégios> **on** <nome-relação ou nome-view> **from** <lista-usuários/papéis>

Portanto, para revogar os privilégios que foram concedidos anteriormente, escrevemos

revoke select on *agência* **from** *John, Mary*
revoke update (*quantia*) **on** *empréstimo* **from** *John, Mary*

A revogação de privilégios será mais complexa se o usuário do qual o privilégio está sendo revogado tiver concedido o privilégio para outro usuário. Retornaremos a essa questão na seção "Autorização em SQL" do Capítulo 8.

SQL embutida

A SQL fornece uma linguagem de consulta declarativa poderosa. Escrever consultas em SQL normalmente é muito mais fácil do que codificar as mesmas consultas em uma linguagem de programação de finalidade geral. Entretanto, um programador precisa ter acesso a um banco de dados por meio de uma linguagem de programação de finalidade geral por, pelo menos, duas razões:

1. Nem todas as consultas podem ser expressas em SQL, já que esta não fornece toda a capacidade expressiva de uma linguagem de finalidade geral. Ou seja, existem consultas que podem ser expressas em uma linguagem como C, Java ou Cobol e que não podem ser expressas em SQL. Para escrevê-las, podemos incorporar SQL dentro de uma linguagem mais poderosa.

   A SQL é projetada de modo que as consultas escritas nela possam ser otimizadas automaticamente e executadas de maneira eficiente – e fornecer toda a capacidade de uma linguagem de programação torna a otimização automática excessivamente difícil.
2. Ações não declarativas – como imprimir um relatório, interagir com um usuário ou enviar os resultados de uma consulta para uma interface gráfica – não podem ser tomadas de dentro da SQL. As apli-

cações normalmente possuem vários componentes, e consultar ou atualizar dados é apenas um componente; outros componentes são escritos em linguagens de programação de finalidade geral. Para uma aplicação integrada, os programas escritos na linguagem de programação precisam ser capazes de acessar o banco de dados.

O padrão SQL define incorporações da SQL em diversas linguagens de programação, como C, Cobol, Pascal, Java, PL/I e Fortran. Uma linguagem em que consultas SQL são embutidas é chamada de linguagem *host*, e as estruturas SQL permitidas na linguagem host constituem a SQL *embutida*.

Os programas escritos na linguagem host podem usar a sintaxe da SQL embutida para acessar e atualizar dados armazenados em um banco de dados. Essa forma embutida da SQL estende ainda mais a capacidade do programador de manipular o banco de dados. Na SQL embutida, todo o processamento de consulta é realizado pelo sistema de banco de dados, que, então, torna o resultado da consulta disponível ao programar uma tupla (registro) de cada vez.

Um programa SQL embutido precisa ser processado por um pré-processador especial antes da compilação. O pré-processador substitui requisições SQL por declarações e chamadas de procedimento da linguagem host, que permitem a execução dos acessos a banco de dados em tempo de execução. Depois, o programa resultante é compilado pelo compilador da linguagem host. Para identificar requisições SQL embutidas para o pré-processador, usamos a instrução EXEC SQL, que tem a forma

EXEC SQL <instrução SQL embutida> END-EXEC

A sintaxe exata para requisições SQL embutidas depende da linguagem em que a SQL está embutida. Por exemplo, um ponto-e-vírgula é usado no lugar de END-EXEC quando a SQL está embutida em C. A incorporação da SQL em Java (chamada SQLJ) usa a sintaxe

SQL { <instrução SQL embutida> };

Colocamos a instrução SQL INCLUDE no programa para identificar o local em que o processador deve inserir as variáveis especiais usadas para comunicação entre o programa e o sistema de banco de dados. As variáveis da linguagem host podem ser usadas dentro de instruções SQL embutidas, mas elas precisam ser precedidas de um sinal de dois-pontos para distingui-las das variáveis SQL.

Antes de executar qualquer instrução SQL, o programa precisa se conectar ao banco de dados. Isso é feito usando

EXEC SQL **connect to** *servidor* **user** *nome-usuário*
END-EXEC

Aqui, *servidor* identifica o servidor com o qual uma conexão deve ser estabelecida. As implementações de banco de dados podem exigir que uma senha seja fornecida além de um nome de usuário.

As instruções SQL embutidas são semelhantes na forma às instruções SQL que descrevemos neste capítulo. Existem, no entanto, várias diferenças importantes, como notamos aqui.

Para escrever uma consulta relacional, usamos a instrução **declare cursor**. O resultado da consulta ainda não é calculado. Em vez disso, o programa precisa usar os comandos **open** e **fetch** (discutidos mais adiante nesta seção) para obter as tuplas resultado.

Considere o esquema de banco que usamos neste capítulo. Considere que temos uma variável de linguagem host *quantia* e que desejamos encontrar os nomes e cidades dos clientes que possuem mais do que o valor de *quantia* em qualquer conta. Podemos escrever essa consulta desta forma:

```
EXEC SQL
  declare c cursor for
  select nome_cliente, cidade_cliente
  from depositante, cliente, conta
  where depositante.nome_cliente = cliente.nome_cliente and
conta.número_conta = depositante.número_conta and
      conta.saldo > :quantia
END-EXEC
```

A variável *c* na expressão anterior é chamada de *cursor* para a consulta. Usamos essa variável para identificar a consulta na instrução **open** (o que faz com que a consulta seja avaliada) e na instrução **fetch** (o que faz com que os valores da tupla sejam colocados em variáveis da linguagem host).

A instrução open para nossa consulta de exemplo é a seguinte:

EXEC SQL **open** *c* END-EXEC

Essa instrução faz com que o sistema de banco de dados execute a consulta e salve os resultados dentro de uma relação temporária. A consulta possui uma variável de linguagem host (*:quantia*) e utiliza o valor da variável no momento em que a instrução **open** é executada.

Se a consulta SQL resultar em um erro, o sistema de banco de dados armazena um diagnóstico de erro nas variáveis da área de comunicação SQL (SQLCA), cujas declarações são inseridas pela instrução SQL INCLUDE.

Um programa SQL embutido executa uma série de instruções **fetch** para recuperar tuplas do resultado. A instrução **fetch** requer uma variável de linguagem host para cada atributo da relação resultado. Para nossa consulta de exemplo, precisamos de uma variável para armazenar o valor *nome_cliente* e outra para armazenar o valor *cidade_cliente*. Suponha que essas variáveis sejam *nc* e *cc*, respectivamente. Então, a instrução

EXEC SQL **fetch** *c* **into** :*nc*, :*cc* END-EXEC

produz uma tupla da relação resultado. O programa pode, então, manipular as variáveis *nc* e *cc* usando os recursos da linguagem de programação nativa.

Uma única requisição **fetch** retorna apenas uma tupla. Para obter todas as tuplas do resultado, o programa precisa conter um loop a fim de iterar sobre todas as tuplas. A SQL embutida auxilia o programador na manutenção dessa iteração. Embora uma relação seja, conceitualmente, um conjunto, as tuplas do resultado de uma consulta estão em alguma ordem física fixa. Quando o programa executa uma instrução **open** em um cursor, o cursor é definido para apontar para a primeira tupla do resultado. Cada vez que ele executa uma instrução **fetch**, o cursor é atualizado para apontar para a próxima tupla do resultado. Quando não houver mais tuplas a serem processadas, a variável SQLSTATE na SQLCA é definida em '02000' (significando "nenhum dado"). Portanto, podemos usar um loop **while** (ou loop equivalente) para processar cada tupla do resultado.

Precisamos usar a instrução **close** para dizer ao sistema de banco de dados para excluir a relação temporária que mantinha o resultado da consulta. Para nosso exemplo, essa instrução tem a forma

EXEC SQL **close** *c* END-EXEC

A incorporação Java da SQL (SQLJ) fornece uma variação desse esquema, em que repetidores são usados no lugar de cursores. A SQLJ associa os resultados de uma consulta com um repetidor, e o método `next( )` da interface repetidora Java pode ser usado para percorrer as tuplas resultado, exatamente como os exemplos anteriores usam **fetch** no cursor.

As expressões SQL embutidas para modificação de banco de dados (**update**, **insert** e **delete**) não retornam um resultado. Portanto, elas são um pouco mais simples de expressar. Uma requisição de modificação de banco de dados tem a forma

EXEC SQL < qualquer **update**, **insert** ou **delete** válido >
END-EXEC

As variáveis da linguagem host, precedidas de um sinal de dois-pontos, podem aparecer na expressão de modificação de banco de dados SQL. Se uma condição de erro surgir na execução da instrução, um diagnóstico é definido na SQLCA.

As relações de banco de dados também podem ser atualizadas com a ajuda de cursores. Por exemplo, se quisermos adicionar 100 ao atributo *saldo* de cada *conta* cujo nome de agência é "Perryridge", poderíamos declarar um cursor assim:

```
declare c cursor for
select *
from conta
where nome_agência = 'Perryridge'
for update
```

Depois, percorremos as tuplas realizando operações **fetch** no cursor (como ilustrado anteriormente) e, após buscar cada tupla, executamos o seguinte código:

```
update conta
set saldo = saldo + 100
where current of c
```

A SQL embutida permite que um programa de linguagem host acesse o banco de dados, mas ela não fornece assistência alguma para apresentar resultados ao usuário ou para gerar relatórios. A maioria dos produtos de banco de dados comerciais inclui ferramentas para auxiliar os programadores de aplicação a criar interfaces com o usuário e relatórios formatados. O Capítulo 8 descreve como construir aplicações de banco de dados com interfaces com o usuário, especialmente interfaces baseadas na Web.

SQL dinâmica

O componente *SQL dinâmica* da SQL permite que programas construam e submetam consultas SQL em tempo de execução. Por outro lado, as instruções SQL embutidas precisam estar completamente presentes em tempo de compilação; elas são compiladas pelo pré-processador SQL embutido. Usando SQL dinâmica, os programas podem criar consultas SQL como strings em tempo de execução (talvez baseados na entrada do usuário) e podem fazê-las ser executadas imediatamente ou ser *preparadas* para uso subseqüente. Preparar uma instrução SQL dinâmica compila a instrução, e os usos subseqüentes da instrução preparada compreendem a versão compilada.

A SQL define padrões para incorporar chamadas SQL dinâmicas em uma linguagem host, como C. Veja o seguinte exemplo:

```
char * prog-sql = "update conta set saldo = saldo * 1,05
                   where numero_conta = ?";
EXEC SQL prepare prog-din from :prog-sql;
char conta[10] = "A-101";
EXEC SQL execute prog-din using :conta;
```

O programa SQL dinâmico contém um?, que é um marcador de lugar para um valor que é fornecido quando o programa SQL é executado.

Entretanto, essa sintaxe exige extensões para a linguagem ou um pré-processador para a linguagem estendida. Uma alternativa comum é usar uma interface de programa de aplicação para enviar consultas ou atualizações SQL para um sistema de banco de dados e não fazer quaisquer mudanças na linguagem de programação em si.

No restante desta seção, veremos dois padrões para conectar a um banco de dados SQL e realizar consultas e atualizações. Um deles, o ODBC, é uma interface de programa de aplicação originalmente desenvolvida para a linguagem C e subseqüentemente estendida para outras linguagens como C++, C# e Visual Basic. O outro padrão, a JDBC, é uma interface de programa de aplicação para a linguagem Java.

Para entender esses padrões, precisamos entender o conceito de sessões SQL. O usuário ou aplicação se *conecta* a um servidor SQL, estabelecendo uma sessão SQL, executa uma série de instruções e, finalmente, *desconecta* a sessão. Portanto, todas as atividades do usuário ou aplicação estão no contexto de uma sessão SQL. Além dos comandos SQL normais, uma sessão também pode conter comandos para *confirmar* o trabalho realizado na sessão, ou para *reverter* o trabalho.

ODBC

O padrão **Open Database Connectivity** (ODBC) define uma maneira para um programa de aplicação se comunicar com um servidor de banco de dados. O ODBC define uma **interface de programa de aplicação** (API) que as aplicações podem usar para abrir uma conexão com um banco de dados, enviar consultas e atualizações e trazer resultados. Aplicações como interfaces gráficas com o usuário, pacotes de estatística e planilhas podem fazer uso da mesma API ODBC para se conectar a qualquer servidor de banco de dados que aceite ODBC.

Cada sistema de banco de dados com suporte a ODBC fornece uma biblioteca que precisa ser vinculada ao programa cliente. Quando o programa cliente faz uma chamada API ODBC, o código na biblioteca se comunica com o servidor para realizar a ação requisitada e buscar resultados.

A Figura 4.4 mostra um exemplo de código C usando a API ODBC. A primeira etapa no uso da ODBC para se comunicar com um servidor é configurar uma conexão com o servidor. Para isso, o programa primeiramente aloca um ambiente SQL e, depois, um descritor de conexão de banco de dados. O ODBC define os tipos `HENV`, `HDBC` e `RETCODE`. Em seguida, o programa abre a conexão de banco de dados usando `SQLConnect`. Essa chamada usa vários parâmetros, incluindo o descritor de conexão, o servidor ao qual a conexão está sendo feita, o identificador de usuário e a senha para o banco de dados. A constante `SQL_NTS` indica que o argumento anterior é uma string terminada por caractere nulo.

Uma vez configurada a conexão, o programa pode enviar comandos para o banco de dados usando `SQLExecDirect`. As variáveis de linguagem C podem ser associadas a atributos do resultado da consulta, de modo que, quando uma tupla resultado é buscada usando `SQLFetch`, seus valores de atributo são armazenados em variáveis C correspondentes. A função `SQLBindCol` desempenha essa tarefa; o segundo argumento identifica a posição do atributo no resultado da consulta, e o terceiro argumento indica a conversão de tipo de SQL para C necessária. O próximo argumento fornece o endereço da variável. Para tipos de tamanho variável como arrays, os dois últimos argumentos fornecem o tamanho máximo da variável e um local onde o tamanho real deve ser armazenado quando uma tupla é buscada. Um valor negativo retornado para o campo length indica que o valor é **nulo**. Para tipos de tamanho fixo como inteiro ou ponto flutuante, o campo maximum length é ignorado, enquanto um valor negativo retornado para o campo length indica um valor nulo.

A instrução `SQLFetch` está em um loop **while**, que é executado até que `SQLFetch` retorne um valor diferente de SQL_SUCCESS. Na busca, o programa armazena os valores em variáveis C como especificado pelas chamadas em SQLBindCol e imprime esses valores.

No final da sessão, o programa libera o descritor de instrução, desconecta-se do banco de dados e libera a conexão e os descritores de ambiente. O bom estilo de programação exige que o resultado de toda chamada de função seja verificado para garantir que não haja qualquer erro; omitimos a maioria dessas verificações por brevidade.

É possível criar uma instrução SQL com parâmetros; por exemplo, considere a instrução `insert into account values (?,?,?)`. Os pontos de interrogação são marcadores de lugar para valores que serão fornecidos mais tarde. Essa instrução pode ser "preparada", ou seja, compilada no banco de dados, e executada repetidamente fornecendo valores reais para os marcadores de lugar – nesse caso, fornecendo um número de conta, um nome de agência e um saldo para a relação *account*.

A ODBC define funções para várias tarefas, tais como encontrar todas as relações no banco de dados e encontrar os nomes e tipos de colunas de um resultado de consulta ou uma relação no banco de dados.

```
void ODBCexample( )
{
    RETCODE error;
    HENV env; /* ambiente */
    HDBC conn; /* conexão a banco de dados */

    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS
               "avipasswd", SQL_NTS);
    {
        char branchname[80];
        float balance;
        int lenOut1, lenOut2;
        HSTMT stmt;

        char * sqlquery = "select branch_name, sum (balance)
                           from account
                           group by branch_name";
        SQLAllocStmt(conn, &stmt);
        error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
        if (error == SQL_SUCCESS) {
            SQLBindCol(stmt, 1, SQL_C_CHAR, branchname, 80, &lenOut1);
            SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance, 0 , &lenOut2);
            while (SQLFetch(stmt) == SQL_SUCCESS) {
                printf ("%s %g\n", branchname, balance);
            }
        }
        SQLFreeStmt(stmt, SQL_DROP);
    }
    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

Figura 4.4 Exemplo de código ODBC.

Como padrão, cada instrução SQL é tratada como uma transação separada que é confirmada automaticamente. A instrução `SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)` desativa a confirmação automática na conexão `conn`, e as transações precisam ser confirmadas explicitamente por `SQLTransact(conn, SQL_COMMIT)` ou revertidas por `SQLTransact(conn, SQL_ROLLBACK)`.

O padrão ODBC define *níveis de conformidade*, que especificam subconjuntos da funcionalidade definida pelo padrão. Uma implementação ODBC pode fornecer apenas recursos de nível básico, ou pode fornecer recursos mais avançados (nível 1 ou nível 2). O nível 1 requer suporte para buscar informações sobre o catálogo, como informações sobre que relações estão presentes e os tipos de seus atributos. O nível 2 requer recursos extras, como a capacidade de enviar e recuperar arrays de valores de parâmetro e recuperar informações de catálogo mais detalhadas.

O padrão SQL define uma **interface em nível de chamada (CLI)**, que é semelhante à interface ODBC. As APIs ADO e ADO.NET são alternativas à ODBC, projetadas para as linguagens Visual Basic e C#; veja as notas bibliográficas para obter mais informações.

JDBC

O padrão **JDBC** define uma API que os programas Java podem usar para se conectar a servidores de banco de dados. (A palavra JDBC era originalmente uma abreviatura para **Java Database Connectivity**, mas a forma completa não é mais usada.)

Abrindo uma conexão e executando consultas

A Figura 4.5 mostra um exemplo de programa Java que usa a interface JDBC. O programa precisa, primeiro, abrir uma conexão com um banco de dados e pode, então, executar instruções SQL. Entretanto, antes de abrir uma conexão, ele carrega os drivers apropriados para o banco de dados usando

`Class.forName`. O primeiro parâmetro para a chamada `getConnection` especifica o nome da máquina em que o servidor é executado (em nosso exemplo, `db.yale.edu`) e o número da porta que ele usa para comunicação (em nosso exemplo, 2000). O parâmetro também especifica que esquema no servidor deve ser usado (em nosso exemplo, `bankdb`), já que um servidor de banco de dados pode aceitar vários esquemas. O primeiro parâmetro também especifica o protocolo a ser usado para a conexão com o banco de dados (em nosso exemplo, `jdbc:oracle:thin:`). Note que o JDBC especifica apenas a API, não o protocolo de comunicação. Um driver JDBC pode aceitar vários protocolos e precisamos especificar um que seja aceito tanto pelo banco de dados quanto pelo driver. Os outros dois argumentos para `getConnection` são um identificador de usuário e uma senha.

Em seguida, o programa cria um descritor de instrução na conexão e o usa para executar uma instrução SQL e trazer resultados. Em nosso exemplo, `stmt.executeUpdate` executa uma instrução update. A construção `try { ... } catch { ... }` permite capturar quaisquer exceções (condições de erro) que surgirem quando chamadas JDBC forem feitas e imprimir uma mensagem apropriada para o usuário.

O programa pode executar uma consulta usando `stmt.executeQuery`. Ele pode recuperar o conjunto de linhas no resultado em um `ResultSet` e buscá-las uma tupla de cada vez usando a função `next( )` no conjunto resultado. A Figura 4.5 mostra duas maneiras de recuperar os valores dos atributos em uma tupla: usando o nome do atributo (*branch_name*) e usando a posição do atributo (2, para indicar o segundo atributo).

A conexão é fechada no final do procedimento. Observe que é importante fechá-la porque existe um limite imposto sobre o número de conexões com o banco de dados; conexões abertas podem exceder esse limite, fazendo com que a aplicação não possa abrir mais conexões com o banco de dados.

Instruções preparadas

Podemos criar uma instrução preparada em que alguns valores são substituídos por "?", especificando, assim, que os valores reais serão fornecidos mais tarde. Alguns sistemas de banco de dados compilam a consulta quando esta é preparada; toda vez que a consulta é executada (com novos va-

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try
    {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:2000:bankdb",
                userid, passwd);
        Statement stmt = conn.createStatement( );
        try {
            stmt.executeUpdate(
                "insert into account values('A-9732', 'Perryridge', 1200)");
        } catch (SQLException sqle)
        {
            System.out.println("Could not insert tuple. "+ sqle);
        }
        ResultSet rset = stmt.executeQuery(
                "select branch_name, avg (balance)
                from account
                group by branch_name");
        while (rset.next( )) {
            System.out.println(rset.getString("branch_name") + "" + rset.getFloat(2));
        }
        stmt.close( );
        conn.close( );
    }
    catch (SQLException sqle)
    {
        System.out.println("SQLException : " + sqle);
    }
}
```

Figura 4.5 Um exemplo de código JDBC.

```
PreparedStatement pStmt = conn.prepareStatement(
                "insert into account values(?,?, ?)");
pStmt.setString(1, "A-9732");
pStmt.setString(2, "Perryridge"); pStmt.setInt(3, 1200);
pStmt.executeUpdate();
pStmt.setString(1, "A-9733");
pStmt.executeUpdate();
```

Figura 4.6 Instruções preparadas em código JDBC.

lores), o banco de dados pode reutilizar a forma anteriormente compilada da consulta. O fragmento de código na Figura 4.6 mostra como instruções preparadas podem ser usadas. A função `setString` (e outras funções semelhantes para outros tipos SQL básicos) permite especificar os valores para os parâmetros.

As instruções preparadas são o método preferido de executar consultas SQL, quando a consulta usa valores inseridos por um usuário. Suponha que os valores para as variáveis `account_number` , `branch_name` e `balance` tenham sido inseridos por um usuário, e uma linha correspondente deve ser inserida na relação *account*. Suponha que, em vez de usar uma instrução preparada, uma consulta seja executada concatenando as strings desta maneira:

```
"insert into account values('" + account_number + "','"
  + branch_name + "'," + balance + ")"
```

e a consulta é executada diretamente. Agora, se o usuário digitasse um apóstrofo nos campos número de conta ou nome agência, a string de consulta teria um erro de sinalização. É bastante provável que uma agência bancária possa ter um apóstrofo em seu nome (especialmente se for um nome irlandês, como O'Henry). Pior ainda, hackers maliciosos podem "injetar" consultas SQL por conta própria digitando caracteres apropriados na string. Essa injeção de SQL pode resultar em sérias brechas de segurança.

Acrescentar caracteres de escape para lidar com caracteres de apóstrofo é uma maneira de resolver esse problema. Usar instruções preparadas é um modo mais simples de resolver o problema, já que o método `setString` acrescenta caracteres de escape implicitamente. Além disso, quando a mesma instrução precisar ser executada várias vezes com valores diferentes, as instruções preparadas normalmente são executadas muito mais rápido dos que instruções SQL separadas.

A JDBC também fornece uma interface `CallableStatement` que permite a chamada de procedimentos armazenados e funções (descritos mais adiante, na seção "Funções e construções procedurais"). Esses desempenham o mesmo papel para funções e procedimentos que `prepareStatement` desempenha para consultas.

```
CallableStatement cStmt1 = conn.prepareCall("{? = call
    some_function(?)}");
CallableStatement cStmt2 = conn.prepareCall("{call
    some_procedure(?,?)}");
```

Os tipos de dados dos valores de retorno de função e os parâmetros de saída dos procedimentos precisam ser registrados usando o método `registerOutParameter( )` e podem ser recuperados usando métodos get semelhantes àqueles para conjuntos resultado. Veja o manual da JDBC para saber mais detalhes.

Recursos de metadados

A JDBC também fornece mecanismos para examinar esquemas de banco de dados e para encontrar os tipos de atributos de um conjunto resultado. A interface `ResultSet` possui um método `getMetaData( )` para obter um objeto `ResultSetMetaData` fornecendo metadados sobre o conjunto resultado. A interface `ResultSetMetaData`, por sua vez, possui métodos para encontrar informações de metadados, como o número de colunas no resultado, o nome de uma coluna especificada ou o tipo de uma coluna especificada. O programa JDBC a seguir ilustra o uso da interface `ResultSetMetaData` para imprimir os nomes e tipos de todas as colunas de um conjunto resultado. As variáveis `rs` no código são consideradas um conjunto resultado obtido executando-se uma consulta.

```
ResultSetMetaData rsmd = rs.getMetaData( );
for(int i = 1; i <= rsmd.getColumnCount( ); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

A interface `DatabaseMetaData` fornece um modo de encontrar metadados sobre o banco de dados. A interface `Connection` possui um método getMetaData que retorna um objeto de `DatabaseMetaData`. A interface `DatabaseMetaData`, por sua vez, possui um grande número de métodos para obter metadados sobre o banco de dados. O código na Figura 4.7 ilustra como encontrar informações sobre colunas (atributos) ou relações em um banco de dados. A variável `conn` é considerada para armazenar uma conexão de banco de dados já aberta. O método `getColumns` usa quatro argumentos:

um nome de catálogo (nulo significa que o nome de catálogo deve ser ignorado), um padrão de nome de esquema, um padrão de nome de tabela e um padrão de nome de coluna. Os padrões de nome de esquema, nome de tabela e nome de coluna podem ser usados para especificar um nome ou um padrão. Os padrões podem usar os caracteres especiais de correspondência de string SQL "%" e "_"; por exemplo, o padrão "%" encontra todos os nomes.

Apenas as colunas das tabelas que satisfazem o nome ou padrão especificado são recuperadas. Cada linha no conjunto resultado contém informações sobre uma coluna. As linhas possuem várias colunas, tais como o nome do catálogo, esquema, tabela e coluna, o tipo da coluna e assim por diante.

Outros métodos fornecidos pela interface `DatabaseMetaData` permitem a recuperação de metadados sobre relações (`getTables( )`), referências de chave estrangeira (`getCrossReference( )`), autorizações, limites de banco de dados, como número máximo de conexões e assim por diante.

As interfaces de metadados podem ser usadas para diversas tarefas. Por exemplo, elas podem ser usadas para escrever um navegador de banco de dados que permita ao usuário encontrar as tabelas em um banco de dados, examinar seu esquema, examinar linhas em uma tabela, aplicar seleções para ver linhas desejadas e assim por diante. As informações de metadados podem ser usadas para tornar genérico o código usado para essas tarefas; por exemplo, o código para exibir as linhas em uma relação pode ser escrito de maneira que funcione em todas as relações possíveis independentemente do seu esquema. Da mesma forma, é possível escrever código que tome uma string de caractere, execute a consulta e imprima os resultados como uma tabela formatada; o código funciona seja qual for a consulta real submetida.

Outros recursos

A JDBC fornece vários outros recursos, como os **conjuntos resultado atualizáveis**. Ele pode criar um conjunto resultado atualizável de uma consulta que realiza uma seleção e/ou uma projeção sobre uma relação de banco de dados. Uma atualização em uma tupla no conjunto resultado, então, resulta em uma atualização na tupla correspondente da relação de banco de dados.

Como padrão, cada instrução SQL é tratada como uma transação separada que é confirmada automaticamente. O método `setAutoCommit( )` na interface JDBC `Connection` permite que esse comportamento seja ativado ou desativado. Portanto, se `conn` for uma variável armazenando uma conexão aberta, `conn.setAutoCommit(false)` desativará a confirmação automática. As transações, então, precisam ser confirmadas explicitamente por `conn.commit( )` ou revertidas por `conn.rollback( )`. Uma confirmação automática pode ser ativada por `conn.setAutoCommit(true)`.

A JDBC fornece interfaces para lidar com objetos grandes sem a necessidade de que um objeto grande inteiro seja criado na memória. Para buscar objetos grandes, a classe `ResultSet` fornece métodos `getBlob( )` e `getClob( )` que são semelhantes ao método `getString( )`, mas retornam objetos do tipo `Blob` e `Clob`, respectivamente. Esses objetos não armazenam o objeto grande inteiro, mas armazenam localizadores para os objetos grandes. As classes `Blob` e `Clob` fornecem métodos para recuperar um objeto grande em pequenas partes. Elas também permitem que objetos grandes sejam armazenados no banco de dados; elas podem ser associadas a fluxos de dados Java, que são buscados transparentemente e enviados ao banco de dados em pequenas partes, de modo que o objeto inteiro não precise ser criado na memória.

A JDBC também fornece uma classe `RowSet`, que possui os recursos de `ResultSet` além de vários recursos extras. Para obter mais informações sobre a JDBC, consulte as notas bibliográficas no final do capítulo.

Funções e construções procedurais**

A partir da versão SQL:1999, a SQL permite a definição de funções, procedimentos e métodos. Esses podem ser definidos pelo componente procedural da SQL:1999 ou por uma linguagem de programação externa, como Java, C ou

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "bankdb", "account", "%");
   // Arguments to getColumns: Catalog, Schema-pattern, Table-pattern, and
   //              Column-Pattern
   // Returns: One row for each column; row has a number of attributes such as
   //              COLUMN_NAME, TYPE_NAME
while( rs.next( ) {
   System.out.println(rs.getString("COLUMN_NAME"), rs.getString("TYPE_NAME");
}
```

Figura 4.7 Encontrando informações de coluna na JDBC com o uso de DatabaseMetaData.

```
create function contagem_conta (nome_cliente varchar(20))
  returns integer
  begin
  declare contagem_c integer;
    select count(*) into contagem_c
    from depositante
    where depositante.nome_cliente = nome_cliente
  return contagem_c;
  end
```

Figura 4.8 Função definida na SQL.

C++. Estudaremos primeiro as definições na SQL e, depois, veremos como usar definições nas linguagens externas.

Vários sistemas de banco de dados aceitam suas próprias extensões procedurais da SQL, como a PL/SQL no Oracle e a TransactSQL no Microsoft SQL Server. Essas extensões se assemelham à parte procedural da SQL, mas pode haver diferenças significantes na sintaxe e na semântica; veja os manuais do respectivo sistema para obter mais detalhes.

Funções e procedimentos SQL

Suponha que queiramos uma função que, dado o nome de um cliente, retorne a contagem do número de contas pertencentes a esse cliente. Podemos definir a função como mostra a Figura 4.8.

A seguinte função pode ser usada em uma consulta que retorna nomes e endereços de todos os clientes com mais de uma conta:

```
select nome_cliente, rua_cliente, cidade_cliente
from cliente
where contagem_conta (nome_cliente) > 1
```

As funções são particularmente úteis com tipos de dados especializados, como imagens e objetos geométricos. Por exemplo, um tipo de dados de segmento de linha em um banco de dados de mapa pode ter uma função associada que verifica se dois segmentos de linha se sobrepõem, e um tipo de dados de imagem pode ter funções associadas para comparar a semelhança de duas imagens.

Desde a versão SQL:2003, a SQL aceita funções que podem retornar tabelas como resultados; essas funções são chamadas **funções de tabela**. Considere a função definida na Figura 4.9. A função retorna uma tabela contendo todas as contas que uma determinada pessoa possui. Observe que o parâmetro de função é referenciado prefixando-o com o nome da função (*contas_de.nome_cliente*).

A função pode ser usada em uma consulta desta maneira:

```
select *
from table (contas_de ('Smith'))
```

Essa consulta retorna todas as contas pertencentes ao cliente 'Smith'. Nesse caso simples, é fácil escrever essa consulta sem usar funções com valor de tabela. Em geral, contudo, as funções com valor de tabela podem ser imaginadas como **views parametrizadas** que generalizam a noção comum de views permitindo parâmetros.

```
create function contas_de (nome_cliente char(20))
  returns table (
    número_conta char(10),
    nome_agência char(15),
    saldo numeric(12,2))
return table
  (select número_conta, nome_agência, saldo
  from conta
  where exists (
    select *
    from depositante
```

Figura 4.9 Função de tabela na SQL.

A SQL:1999 também aceita procedimentos. A função *contagem_conta* poderia ser escrita como um procedimento:

```
create procedure proc_contagem_conta (in nome_cliente
                varchar(20), out contagem_c integer)
  begin
    select count(*) into contagem_c
    from depositante
    where depositante.nome_cliente =
proc_contagem_conta.nome_cliente
  end
```

Os procedimentos podem ser chamados a partir de um procedimento SQL ou de SQL embutida pela instrução **call**:

```
declare contagem_c integer;
call proc_contagem_conta('Smith', contagem_c);
```

Os procedimentos e funções podem ser chamados a partir de SQL dinâmica, como ilustrado pela sintaxe JDBC na seção "Instruções preparadas".

A SQL:1999 permite mais de um procedimento do mesmo nome, desde que o número de argumentos dos procedimentos com o mesmo nome seja diferente. O nome, juntamente com o número de argumentos, é usado para identificar o procedimento. A SQL também permite mais de uma função com o mesmo nome, desde que as diferentes funções com o mesmo nome tenham números diferentes de argumentos ou desde que as funções com o mesmo número de argumentos difiram no tipo de pelo menos um argumento.

Construções procedurais

Desde a versão SQL:1999, a SQL aceita diversas construções procedurais, o que lhe confere quase toda a capacidade de uma linguagem de programação de finalidade geral. A parte do padrão SQL que lida com essas construções é chamada de **Persistent Storage Module (PSM)**.

O objetivo do PSM SQL não é substituir as linguagens de programação convencionais. Em vez disso, as construções procedurais permitem que "lógica empresarial" seja registrada como procedimentos armazenados no banco de dados, e executada dentro do banco de dados. Por exemplo, os bancos normalmente têm muitas regras sobre como e quando um pagamento pode ser feito a um cliente, como limites de saque em dinheiro, requisitos de saldo mínimo, ofertas de crédito que permitem que um cliente saque mais do que o saldo disponível gerando automaticamente um empréstimo, e assim por diante.

Embora essa lógica empresarial possa ser codificada como procedimentos de linguagem de programação armazenados inteiramente fora do banco de dados, defini-las como procedimentos armazenados no banco de dados oferece várias vantagens. Por exemplo, isso permite que várias aplicações acessem os procedimentos e possibilita um único ponto de mudança no caso de as regras empresariais mudarem, sem alterar a aplicação. O código de aplicação, então, pode chamar procedimentos armazenados, em vez de atualizar diretamente relações de banco de dados.

As construções procedurais são necessárias para permitir que regras empresariais complexas sejam codificadas como procedimentos armazenados; por isso, elas foram incluídas na SQL a partir da versão SQL:1999 (elas eram aceitas por alguns produtos de banco de dados mesmo antes disso).

Uma instrução composta tem a forma **begin** ... **end**, e pode conter várias instruções SQL entre o **begin** e o **end**. Variáveis locais podem ser declaradas dentro de uma instrução composta, como vimos na seção anterior.

A SQL:1999 aceita as instruções **while** e as instruções **repeat** seguindo a sintaxe:

```
declare n integer default 0;
while n < 10 do
  set n = n + 1;
end while
repeat
  set n = n – 1;
until n = 0
end repeat
```

Esse código não faz nada de útil; ele simplesmente mostra a sintaxe dos loops **while** e **repeat**. Veremos usos mais significativos posteriormente.

Existe também um loop **for** que permite interação sobre todos os resultados de uma consulta:

```
declare n integer default 0;
for r as
    select saldo from conta
    where nome_agência = 'Perryridge'
do
  set n = n + r.saldo
end for
```

O programa implicitamente abre um cursor quando o loop for inicia a execução e o usa para buscar os valores uma linha de cada vez para a variável do loop **for** (*r*, no exemplo anterior). É possível dar um nome ao cursor, inserindo o texto *nc* cursor for imediatamente após a palavra-chave as, onde *nc* é o nome que queremos dar ao cursor. O nome do cursor pode ser usado para realizar operações de atualização/exclusão na tupla sendo apontada pelo cursor. A instrução leave pode ser usada para sair do loop.

enquanto **iterate** inicia na próxima tupla, desde o início do loop, saltando as instruções restantes.

As instruções condicionais aceitas pela SQL incluem instruções if-then-else usando esta sintaxe:

```
if r.saldo < 1000
  then set l = l + r.saldo
elseif r.saldo < 5000
  then set m = m + r.saldo
else set h = h + r.saldo
end if
```

Esse código considera que *l*, *m* e *h* são variáveis de inteiro e *r* é uma variável de linha. Se substituirmos a linha "**set** *n = n + r.saldo*" no loop **for** do parágrafo anterior pelo código **if-then-else** (e forneceremos declarações e valores iniciais apropriados para *l*, *m* e *h*), o loop calculará os saldos totais das contas que entram nas categorias de saldo baixo, médio e alto, respectivamente.

A SQL aceita uma instrução case semelhante à instrução case da linguagem C/C++ (além das expressões case, que vimos no Capítulo 3).

Finalmente, a SQL inclui o conceito de sinalizar **condições de exceção** e declarar **handlers** que possam manipular a exceção, como neste código:

```
declare fora_de_estoque condition
declare exit handler for fora_de_estoque
begin
...
end
```

As instruções entre o begin e o end podem gerar uma exceção executando signal *fora_de_estoque*. O handler diz que se a condição surgir, a ação a ser tomada é sair da instrução begin end. As ações alternativas seriam continue, que continua a execução a partir da próxima instrução seguinte à que gerou a exceção. Além das condições definidas explicitamente, há também condições predefinidas, como sqlexception, sqlwarning e not found.

A Figura 4.10 fornece um exemplo maior do uso das construções procedurais na SQL. A função *saque* definida na figura retira dinheiro de uma conta, e, se o saldo se tornar negativo, ela inicia a manipulação do saque a descoberto; o código para manipular saque a descoberto não é mostrado. A função retorna um código de erro, com um valor maior ou igual a 0 significando sucesso e um valor negativo significando uma condição de erro.

Outro exemplo que ilustra loops while é apresentado mais adiante, na seção "Consultas recursivas".

```
create procedure saque(
        in número_conta varchar(10)
        in quantia numeric(12,2))
- - retira dinheiro de uma conta
returns integer
begin
  declare novosaldo numeric(12,2);
  select saldo into novosaldo
  from conta
  where conta.número_conta = saque.número_conta;
  novosaldo = novosaldo – quantia;
  if (novosaldo < 0)
    begin
      ... código para manipular saque a descoberto aqui
    ... se quantia muito grande para ser manipulada por saque a descoberto, código de erro -1
    end
  else begin
  update conta
    set saldo = saldo – novosaldo
    where conta.número_conta = saque.número_conta
  end
  return(0);
end
```

Figura 4.10 Procedimento para saque da conta.

Rotinas externas à linguagem

A SQL permite definir funções em uma linguagem de programação como Java, C#, C ou C++. As funções definidas dessa maneira podem ser mais eficientes do que as funções definidas na SQL, e os cálculos que não podem ser realizados na SQL podem ser executados por essas funções. Um exemplo do uso dessas funções seria realizar um cálculo aritmético complexo nos dados em uma tupla.

Os procedimentos e funções externos podem ser especificados desta maneira:

```
create procedure proc_contagem_conta(in nome_cliente
                                     varchar(20), out contagem integer)
language C
external name '/usr/avi/bin/proc_contagem_conta'

create function contagem_conta (nome_cliente
varchar(20))
returns inteiro
language C
external name '/usr/avi/bin/contagem_conta'
```

Os procedimentos de linguagem externa precisam lidar com valores nulos e exceções. Portanto, eles precisam ter vários parâmetros: um valor **sqlstate** para indicar estado de falha/sucesso, um parâmetro para armazenar o valor de retorno da função, e variáveis de indicador para cada resultado de parâmetro/função a fim de indicar se o valor é nulo. Uma linha extra **parameter style general** acrescentada à declaração anterior indica que os procedimentos/funções tomam apenas os argumentos mostrados e não lidam com valores nulos ou exceções.

As funções definidas em uma linguagem de programação e compiladas fora do sistema de banco de dados podem ser carregadas e executadas com o código do sistema de banco de dados. Entretanto, isso causa o risco de que um bug no programa possa danificar as estruturas internas do banco de dados e possa evitar a funcionalidade de controle de acesso do sistema de banco de dados. Os sistemas de banco de dados que se preocupam mais com um desempenho eficiente do que com a segurança podem executar procedimentos dessa maneira. Os sistemas de banco de dados que se preocupam mais com a segurança podem executar esse código como parte de um processo separado, comunicar os valores de parâmetro para ele e buscar os resultados com a comunicação interprocessos. Contudo, o overhead de tempo da comunicação interprocessos é muito alto; em arquiteturas de CPU comuns, dezenas a centenas de milhares de instruções podem ser executadas no tempo gasto para uma comunicação interprocessos.

Se o código for escrito em uma linguagem "segura", como Java ou C#, existe outra possibilidade: executar o código em uma **sandbox** dentro do próprio processo de execução da consulta de banco de dados. A sandbox permite que o código Java ou C# acesse sua própria área de memória, mas impede que o código leia ou atualize a memória no processo de execução da consulta, ou que acesse arquivos no sistema de arquivos. (A criação de uma sandbox não é possível para uma linguagem como C, que permite acesso irrestrito à memória por meio de ponteiros.)

Vários sistemas de banco de dados atualmente aceitam rotinas externas à linguagem em uma sandbox dentro do processo de execução da consulta. Por exemplo, o Oracle e o IBM DB2 permitem que funções Java sejam executadas como parte do processo de banco de dados. O Microsoft SQL Server 2005 permite que procedimentos sejam compilados para a Common Language Runtime (CLR) de modo a serem executados dentro do processo de banco de dados; esses procedimentos poderiam ter sido escritos, por exemplo, em C# ou Visual Basic.

Consultas recursivas**

Considere um banco de dados contendo informações sobre os funcionários de uma organização. Suponha que tenhamos uma relação *gerente*(*nome_funcionário*, *nome_gerente*), especificando qual funcionário é diretamente supervisionado por qual gerente. A Figura 4.11 mostra um exemplo da relação *gerente*.

nome_funcionário	*nome_gerente*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Figura 4.11 Relação *gerente*.

Agora, suponha que queiramos descobrir quais funcionários são supervisionados, direta ou indiretamente, por um determinado gerente – digamos, Jones. Ou seja, queremos encontrar funcionários que sejam diretamente supervisionados por Jones, ou que sejam supervisionados por alguém que é supervisionado por Jones, ou, ainda, que é supervisionado por alguém que é supervisionado por alguém que é supervisionado por Jones, e assim por diante.

Portanto, se o gerente de Alon é Barinsky, o gerente de Barinsky é Estovar, e o gerente de Estovar é Jones, então, Alon, Barinsky e Estovar são os funcionários supervisionados por Jones.

Fechamento transitivo usando repetição

Uma forma de escrever a consulta anterior é usar repetição: primeiro, encontre os funcionários que trabalham diretamente subordinados a Jones, depois, os que trabalham subordinados ao primeiro conjunto, e assim por diante. A Figura 4.12 mostra uma função *encontFunc*(*gte*) para realizar essa tarefa; a função toma o nome do gerente como um parâmetro (*gte*), calcula o conjunto de todos os funcionários diretos ou indiretos desse gerente e retorna o conjunto.

O procedimento usa três tabelas temporárias: *funcionário*, que é usada para armazenar o conjunto de tuplas a ser retornado; *novofunc*, que armazena os funcionários encontrados na repetição anterior; e *temp*, que é usada como armazenamento temporário enquanto os conjuntos de funcionários são manipulados. O procedimento insere todos os funcionários que trabalham diretamente para *gte* em *novofunc* antes do loop **repeat**. O loop **repeat** primeiro inclui em *func* todos os funcionários em *novofunc*. Em seguida, ele calcula os funcionários que trabalham para os que estão em *novofunc*, exceto os que já foram encontrados como sendo funcionários de *gte*, e os armazena na tabela temporária, *temp*. Finalmente, ele substitui o conteúdo de *novofunc* pelo conteúdo de *temp*. O loop **repeat** termina quando não encontra qualquer funcionário novo (indireto).

A Figura 4.13 mostra os funcionários que seriam encontrados em cada repetição, se o procedimento fosse chamado para o gerente Jones.

Podemos observar que o uso da cláusula **except** na função garante que esta funcione mesmo no caso (anormal) em que exista um ciclo de gerência. Por exemplo, se *a* trabalha para *b*, *b* trabalha para *c* e *c* trabalha para *a*, existe um ciclo.

Embora os ciclos possam ser irreais no controle de gerencia, os ciclos são possíveis em outras aplicações. Por exemplo, suponha que tenhamos uma relação *vôos* (*para*, *de*) que diga quais cidades podem ser alcançadas de que outras cidades por um vôo direto. Podemos escrever código semelhante ao código na função *encontFunc* para encontrar todas as cidades que são alcançáveis por uma seqüência de um ou mais vôos de uma determinada cidade. Tudo o que precisamos fazer é substituir *gerente* por *vôo* e substituir os nomes de atributo apropriadamente. Nessa situação, pode haver ciclos de alcance, mas a função agiria corretamente, já que ela eliminaria cidades que já tenham sido vistas.

Recursão na SQL

O **fechamento transitivo** da relação *gerente* é uma relação que contém todos os pares (*func*, *gte*) tais que *func* seja um funcionário direto ou indireto de *gte*. Existem várias aplicações que exigem cálculo de fechamentos transitivos semelhantes em **hierarquias**. Por exemplo, as organizações normalmente consistem em vários níveis de unidades organizacionais. As máquinas consistem em peças que, por sua vez, contêm subpeças e assim por diante; por exemplo, uma bicicleta pode ter subpeças como rodas e pedais, que, por sua vez, possuem subpeças como pneus, aros e raios. O fechamento transitivo pode ser usado nessas hierarquias para encontrar, por exemplo, todas as peças de uma bicicleta.

É bastante inconveniente especificar fechamento transitivo usando repetição. Existe um método alternativo e mais fácil, usando definições de view recursivas.

Podemos usar recursão para definir o conjunto de funcionários controlados por um gerente em especial, digamos, Jones, da seguinte maneira. As pessoas supervisionadas (direta ou indiretamente) por Jones são:

1. Pessoas cujo gerente é Jones
2. Pessoas cujo gerente é supervisionado (direta ou indiretamente) por Jones

Observe que o caso 2 é recursivo, já que ele define o conjunto de pessoas supervisionadas por Jones em função do conjunto de pessoas supervisionadas por Jones. Outros exemplos de fechamento transitivo, como encontrar todas as subpeças (diretas ou indiretas) de uma determinada peça também podem ser definidos de um modo semelhante, recursivamente.

Desde a versão SQL:1999, o padrão SQL aceita uma forma limitada de recursão, usando a cláusula **with recursive**, onde uma view (ou view temporária) é expressa em função de si mesma. As consultas recursivas podem ser usadas, por exemplo, para expressar fechamento transitivo concisamente. Lembre-se de que a cláusula **with** é usada para definir uma view temporária cuja definição está disponível apenas para a consulta em que ela está definida. A palavra-chave adicional **recursive** especifica que a view é recursiva.

Por exemplo, podemos encontrar todos os pares (*func*, *gte*) tais que *func* seja direta ou indiretamente gerenciado por *gte*, usando a view SQL recursiva mostrada na Figura 4.14.

```
create function encontFunc(gte char(10))
  - - Encontra todos os funcionários que trabalham direta ou
  - - indiretamente para gte.
returns table (nome char(10))
  - - A relação gerente(nome_funcionário, nome_gerente) especifica quem
  - - trabalha diretamente para quem.
begin
  create temporary table func (nome char(10));
    - - tabela func armazena o conjunto de funcionários a ser retornado
  create temporary table novofunc (nome char(10));
    - - tabela novofunc contém funcionários encontrados na repetição anterior.
  create temporary table temp (nome char(10));
    - - tabela temp é uma tabela temporária usada para armazenar
    - - resultados intermediários.
  insert into novofunc
    select nome_funcionário
    from gerente
    where nome_gerente = gte
  repeat
    insert into func
      select nome
      from novofunc;

    insert into temp
      (select gerente.nome_funcionário
        from novofunc, gerente
        where novofunc.nome_funcionário = gerente.nome_gerente;
      )
      except (
        select nome_funcionário
        from func
      );
    delete from novofunc;
    insert into novofunc
      select *
      from temp;
    delete from temp;

  until not exists (select * from novofunc)
  end repeat;
  return table func
end
```

Figura 4.12 Encontrando todos os funcionários de um gerente.

Número da repetição	*Tuplas em func*
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Anon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Anon)

Figura 4.13 Funcionários de Jones em repetições da função *encontFunc*.

```
with recursive func (nome_funcionário, nome_gerente) as (
    select nome_funcionário, nome_gerente
    from gerente
  union
    select gerente.nome_funcionário, func.nome_gerente
    from gerente, func
    where gerente.nome_gerente = func.nome_funcionário
  )
select *
from func
```

Figura 4.14 Consulta recursiva na SQL.

Qualquer view recursiva precisa ser definida como a união de duas subconsultas: uma **consulta base**, que não é recursiva, e uma **consulta recursiva**, que usa a view recursiva. No exemplo da Figura 4.14, a consulta base é a seleção em *gerente*, enquanto a consulta recursiva calcula a junção de *gerente* e *func*.

O significado de uma view recursiva é mais bem entendido da seguinte forma. Primeiro, calcule a consulta base e acrescente todas as tuplas resultantes à relação de view (que inicialmente está vazia). Em seguida, calcule a consulta recursiva usando o conteúdo atual da relação de view, e acrescente todas as tuplas resultantes novamente à relação de view. A instância da relação de view resultante é chamada de um **ponto fixo** da definição de view recursiva. (O termo "fixo" se refere ao fato de que não há mais mudanças.) A relação de view, portanto, é definida para conter exatamente as tuplas na instância do ponto fixo.

Aplicando essa lógica a nosso exemplo, primeiro encontraríamos todos os funcionários diretos de cada gerente executando a consulta base. A consulta recursiva acrescentaria mais um nível de funcionários em cada repetição, até que seja atingida a profundidade máxima da relação gerente-funcionário. Nesse ponto, nenhuma nova tupla seria incluída na view, e a repetição teria atingido um ponto fixo.

Note que o sistema de banco de dados não precisa usar essa técnica de repetição para calcular o resultado da consulta recursiva; ele pode obter o mesmo resultado usando outras técnicas que possam ser mais eficientes.

Existem algumas restrições na consulta recursiva em uma view recursiva; especificamente, a consulta deve ser **monotônica**, isto é, seu resultado em uma instância de relação de view V_1 deve ser um superconjunto do seu resultado em uma instância de relação de view V_2 se V_1 for um superconjunto de V_2. Intuitivamente, se mais tuplas forem acrescentadas à relação de view, a consulta recursiva deve retornar pelo menos o mesmo conjunto de tuplas de antes, e, possivelmente, retornar tuplas adicionais.

Em especial, as consultas recursivas não devem usar qualquer uma das seguintes construções, já que elas tornariam a consulta não monotônica:

- Agregação na view recursiva
- **not exists** em uma subconsulta que usa a view recursiva
- Diferença de conjunto (**except**) cujo lado direito usa a view recursiva

Por exemplo, se a consulta recursiva fosse da forma $r - v$, onde v é a view recursiva, se incluirmos uma tupla em v, o resultado da consulta pode se tornar menor; a consulta, portanto, não é monotônica.

O significado das views recursivas pode ser definido pelo procedimento iterativo, contanto que a consulta recursiva seja monotônica; se a consulta recursiva não for monotônica, o significado da view será de difícil definição. Portanto, a SQL exige que as consultas sejam monotônicas. As consultas recursivas são discutidas em mais detalhes no contexto da linguagem de consulta Datalog, na seção "Recursão na Datalog" do Capítulo 5.

A SQL também permite a criação de views permanentes definidas recursivamente usando **create recursive view** no lugar de **with recursive**. Algumas implementações aceitam consultas recursivas usando uma sintaxe diferente; veja os manuais do sistema respectivo para obter mais detalhes.

Recursos SQL avançados**

A linguagem SQL cresceu durante as últimas duas décadas de uma simples linguagem com alguns recursos para uma linguagem bastante complexa, com recursos para satisfazer muitos tipos de usuários diferentes. Abordamos os aspectos básicos da SQL anteriormente neste capítulo. Nesta seção, apresentaremos alguns dos recursos mais complexos da SQL. Muitos desses recursos foram introduzidos em versões relativamente recentes da SQL, e podem não ser aceitos por todos os sistemas de banco de dados.

Extensões create table

As aplicações frequentemente exigem a criação de tabelas que possuem o mesmo esquema de uma tabela existente. A SQL fornece uma extensão **create table** para manipular essa tarefa:

```
create table temp_conta like conta
```

Essa instrução cria uma nova tabela *temp_conta* que tem o mesmo esquema de *conta*.

Ao escrever uma consulta complexa, normalmente é útil armazenar o resultado de uma consulta como uma nova tabela; a tabela geralmente é temporária. Duas instruções são necessárias, uma para criar a tabela (com colunas apropriadas) e a outra para inserir o resultado da consulta na tabela. A SQL:2003 fornece uma técnica mais simples para criar uma tabela contendo os resultados de uma consulta. Por exemplo, a instrução a seguir cria uma tabela *t1* contendo os resultados de uma consulta.

```
create table t1 as
    (select *
    from conta
    where nome_agência = 'Perryridge')
with data
```

Como padrão, os nomes e os tipos de dados das colunas são inferidos do resultado da consulta. Nomes podem ser explicitamente atribuídos às colunas listando-se os nomes de coluna após o nome da relação. Se a cláusula **with data** for omitida, a tabela é criada mas não preenchida com dados.

A instrução **create table ... as** anterior se parece muito com a instrução **create view**, e ambas são definidas usando consultas. A principal diferença é que o conteúdo da tabela é definido quando a tabela é criada, enquanto o conteúdo de uma view sempre reflete o resultado da consulta atual.

Note que várias implementações aceitam a funcionalidade de **create table ... like** e **create table ... as** usando sintaxe diferente; veja os manuais do respectivo sistema para obter mais detalhes.

Mais informações sobre subconsultas

A SQL:2003 permite que subconsultas ocorram sempre que um valor é necessário, desde que a subconsulta retorne apenas um valor; essas subconsultas são chamadas **subconsultas escalares**. Por exemplo, uma subconsulta pode ser usada na cláusula **select** como ilustrado no seguinte exemplo, que lista todos os clientes juntamente com o número das contas que possuem:

```
select nome_cliente,
    (select count(*)
    from conta
    where conta.nome_cliente = cliente.nome_cliente)
        as num-contas
from cliente
```

A subconsulta nesse exemplo com certeza retornará apenas um único valor, já que ela é uma agregada **count**(*) sem um **group by**. Subconsultas sem agregadas também são permitidas. Essas consultas podem retornar mais de uma resposta; nesse caso, um erro de runtime ocorre.

As subconsultas na cláusula **select** da consulta interna podem acessar atributos das relações na cláusula **from** da consulta externa, como *cliente.nome_cliente* no exemplo anterior.

Entretanto, as subconsultas na cláusula **from** (vistas anteriormente na seção "Relações derivadas" do Capítulo 3), normalmente não podem acessar atributos de outras relações na cláusula **from**; a SQL:2003 aceita uma cláusula lateral que permite que uma subconsulta na cláusula **from** acesse atributos de subconsultas anteriores na cláusula **from**. Assim, a consulta acima poderia ser escrita alternativamente desta maneira:

```
select nome_cliente, num_contas
  from cliente,
    lateral(select count(*)
        from conta
        where conta.nome_cliente = cliente.nome_cliente)
      as este_cliente (num-contas)
```

Construções avançadas para atualização de banco de dados

Suponha que temos uma relação *fundos_recebidos* (*numero_conta*, *quantia*) que armazene fundos recebidos (por exemplo, por transferência eletrônica de fundos) para cada conta de um conjunto de contas. Agora, imagine que queremos adicionar as quantias aos saldos das respectivas contas. Para usar a instrução update SQL para realizar essa tarefa, precisamos pesquisar a tabela *fundos_recebidos* para cada tupla na tabela *conta*. Podemos usar subconsultas na cláusula update para fazer isso, como a seguir. Para simplicidade, consideramos que a relação *fundos_recebidos* contém no máximo uma tupla para cada conta.

```
update conta set balance = balance +
        (select quantia
        from fundos-recebidos
        where fundos-recebidos.numero_conta =
conta.numero_conta)
```

```
where exists(
    select *
    from fundos-recebidos
    where fundos-recebidos.número_conta =conta.
      número_conta)
```

Observe que a condição na cláusula **where** da atualização garante que apenas contas com tuplas correspondentes em *fundos-recebidos* são atualizadas, enquanto a subconsulta dentro da cláusula **set** calcula a quantia a ser adicionada a cada uma dessas contas.

Existem muitas aplicações que exigem atualizações como a ilustrada anteriormente. Em geral, existe uma tabela, que chamaremos de **tabela mestra**, e as atualizações na tabela mestra são recebidas como um lote. Agora, a tabela mestra precisa ser igualmente atualizada. A SQL:2003 fornece uma construção especial, chamada construção **merge**, para simplificar a tarefa de realizar essa mesclagem de informações. Por exemplo, a atualização anterior pode ser expressa usando **merge** desta maneira:

```
merge into conta as A
using   (select *
         from fundos_recebidos) as F
    on (A.número_conta = F.número_conta)
when matched then
    update set saldo = saldo+F.quantia
```

Quando um registro de uma subconsulta na cláusula **using** corresponde a um registro na relação *conta*, a cláusula **when matched** é executada, o que pode gerar uma atualização na relação; nesse caso, o registro correspondente na relação *conta* é atualizado como mostramos.

A instrução **merge** também pode ter uma cláusula **when not matched then**, que permite inserção de novos resultados na relação. No exemplo anterior, quando não há qualquer conta correspondente para uma tupla *fundos_recebidos*, a ação de inserção pode criar um novo registro de conta (com um *nome_agência* nulo) usando a seguinte cláusula:

```
when not matched then
    insert values (F.número_conta, nulo, F.quantia)
```

Embora não seja muito significativa nesse exemplo,[2] a cláusula **when not matched** pode ser bastante útil em outros casos. Por exemplo, suponha que a relação local seja uma cópia de uma relação mestra, e recebemos registros atualizados e recém-inseridos dessa relação. A instrução **merge** pode atualizar registros correspondentes (esses seriam registros antigos atualizados) e inserir registros que não são correspondentes (esses seriam registros novos).

Nem todas as implementações SQL atualmente aceitam a instrução **merge**; veja os manuais do respectivo sistema para obter mais detalhes.

[2]Uma ação melhor aqui teria sido inserir esses registros em uma relação de erro, mas isso não pode ser feito com a instrução **merge**.

Resumo

- A linguagem de definição de dados SQL fornece suporte para definir tipos de domínio internos, como data e hora, bem como tipos de domínio definidos pelo usuário.
- As restrições de domínio especificam o conjunto de valores possíveis que podem ser associados a um atributo. Essas restrições também podem proibir o uso de valores nulos para determinados atributos.
- As restrições de integridade referencial garantem que um valor que aparece em uma relação para um determinado conjunto de atributos também apareça para um determinado conjunto de atributos em outra relação.
- Afirmações são expressões declarativas que indicam os predicados que exigimos que sempre sejam verdadeiros.
- Um usuário pode ter várias formas de autorização em partes de um banco de dados. A autorização é um meio pelo qual o sistema de banco de dados pode ser protegido contra acesso malicioso ou não-autorizado.
- As consultas SQL podem ser chamadas a partir de linguagens host, por meio de SQL embutida e dinâmica. Os padrões ODBC e JDBC definem interfaces de programa de aplicação para acessar bancos de dados SQL a partir de programas de linguagem C e Java. Cada vez mais programadores estão usando essas APIs para acessar bancos de dados.
- As funções e procedimentos podem ser definidos usando SQL. Também resumimos as extensões procedurais fornecidas pela SQL:1999, que permitem repetição e instruções condicionais (if-then-else).
- Algumas consultas, como fechamento transitivo, podem ser expressas usando repetição ou usando consultas SQL recursivas. A recursão pode ser expressa usando views recursivas ou definições de cláusula **with** recursivas.
- Também vimos uma breve descrição de alguns recursos avançados da SQL, que simplificam certas tarefas relacionadas à definição de dados e à consulta e atualização de dados.

Termos de revisão

- Tipos definidos pelo usuário
- Domínios
- Objetos grandes
- Catálogos
- Esquemas

- Restrições de integridade
- Restrições de domínio
- Restrição unique
- Cláusula check
- Integridade referencial
- Restrição de chave primária
- Restrição de chave estrangeira
- Exclusão em cascata
- Atualização em cascata
- Afirmação
- Autorização
- Privilégios
  - □ Select
  - □ Insert
  - □ Update
  - □ Delete
  - □ All privileges
- Concessão de privilégios
- Revogação de privilégios
- SQL embutida
- Cursores
- Cursores atualizáveis
- SQL dinâmica
- ODBC
- JDBC
- Instruções preparadas
- Acesso a metadados
- Funções SQL
- Procedimentos armazenados
- Construções procedurais
- Rotinas externas à linguagem
- Consultas recursivas
- Consultas monotônicas
- Instrução merge

Exercícios práticos

4.1 Complete a definição DDL SQL do banco de dados de banco da Figura 4.2 para incluir as relações *empréstimo* e *tomador*.

4.2 Considere o seguinte banco de dados relacional:

funcionário (*nome_funcionário*, *rua*, *cidade*)
trabalha (*nome_funcionário*, *nome_empresa*, *salário*)
empresa (*nome_empresa*, *cidade*)
gerencia (*nome_funcionário*, *nome_gerente*)

Forneça uma definição DDL SQL desse banco de dados. Identifique restrições de integridade referencial que devem se aplicar e inclua-as na definição DDL.

4.3 Escreva condições **check** para os esquemas que você definiu no Exercício 4.2 de modo a garantir que:

a. Todo funcionário trabalhe para uma empresa localizada na mesma cidade onde o funcionário mora.
b. Nenhum funcionário receba um salário maior do que o do seu gerente.

4.4 A SQL permite que uma dependência de chave estrangeira se refira à mesma relação, como no seguinte exemplo:

```
create table gerente
    (nome_funcionário    char(20) not null
     nome_gerente        char(20) not null,
     primary key nome_funcionário,
     foreign key (nome_gerente) references gerente
                 on delete cascade)
```

Aqui, *nome_funcionário* é uma chave para a tabela *gerente*, significando que cada funcionário tenha pelo menos um gerente. A cláusula de chave estrangeira exige que todo gerente também seja um funcionário. Explique o que acontece exatamente quando uma tupla na relação *gerente* é excluída.

4.5 Escreva uma afirmação para o banco de dados de banco de modo a garantir que o valor do ativo para a agência Perryridge seja igual à soma das quantias emprestadas pela agência Perryridge.

4.6 Descreva as circunstâncias em que você escolheria usar SQL embutida em vez de SQL pura ou apenas uma linguagem de programação de finalidade geral.

Exercícios

4.7 As restrições de integridade referencial conforme definidas neste capítulo envolvem exatamente duas relações. Considere um banco de dados que inclui as seguintes relações:

trabalhador_assalariado (**nome**, **escritório**, *telefone*, *salário*)
trabalhador_horário (**nome**, **salário_hora**)
endereço (**nome**, **rua**, **cidade**)

Suponha que desejamos exigir que todo nome que apareça em *endereço* também apareça em *trabalhador_assalariado* ou *trabalhador_horário*, mas não necessariamente em ambos.

a. Proponha uma sintaxe para expressar essas restrições.
b. Discuta as ações que o sistema precisa tomar para impor uma restrição desse tipo.

4.8 Escreva uma função Java usando recursos de metadados JDBC que tomam um `ResultSet` como um

parâmetro de entrada e que imprima o resultado na forma tabular, com nomes apropriados como cabeçalhos de coluna.

4.9 Escreva uma função Java usando recursos de metadados JDBC que imprime uma lista de todas as relações no banco de dados, exibindo para cada relação os nomes e tipos desses atributos.

4.10 Considere um banco de dados de funcionários com duas relações

funcionário (*nome_funcionário*, *rua*, *cidade*)
trabalha (*nome_funcionário*, *nome_empresa*, *salário*)

onde as chaves primárias estão sublinhadas. Escreva uma consulta para encontrar empresas cujos funcionários ganham um salário mais alto, em média, do que o salário médio no First Bank Corporation.

a. Usando funções SQL como apropriado.
b. Sem usar funções SQL.

4.11 Reescreva a consulta na seção "Funções e procedimentos SQL" que retorna o nome, a rua e a cidade de todos os clientes com mais de uma conta, usando a cláusula **with** em vez de usar uma chamada de função.

4.12 Compare o uso da SQL embutida com o uso na SQL das funções definidas em uma linguagem de programação de finalidade geral. Sob que circunstâncias você usaria cada um desses recursos?

4.13 Modifique a consulta recursiva na Figura 4.14 para definir uma relação

profundidade_funcionário (*nome_funcionário*, *nome_gerente*, *profundidade*)

onde o atributo *profundidade* indica quantos níveis de gerentes intermediários existem entre o funcionário e o gerente. Os funcionários que estão diretamente sob um gerente teriam uma profundidade 0.

4.14 Considere o esquema relacional

peça (*id_peça*, *nome*, *custo*)
subpeça (*id_peça*, *id_subpeça*, *contagem*)

Uma tupla $(p_1, p_2, 3)$ na relação *subpeça* indica que a peça com o id_peça p_2 é uma subpeça direta da peça com o id_peça p_1, e p_1 tem 3 cópias de p_2. Observe que p_2 pode, ele mesmo, ter outras subpeças. Escreva uma consulta SQL recursiva que gere os nomes de todas as subpeças da peça com o id_peça "P-100".

4.15 Considere novamente o esquema relacional do Exercício 4.14. Escreva uma função JDBC usando SQL não recursiva para encontrar o custo total da peça "P-100", incluindo os custos de todas as subpeças. Certifique-se de levar em conta o fato de que a peça pode ter várias ocorrências de uma subpeça. Você pode usar recursão em Java se desejar.

Notas bibliográficas

Vejas as notas bibliográficas do Capítulo 3 como referências para os padrões SQL e livros sobre SQL.

Muitos produtos de banco de dados aceitam recursos SQL além dos especificados nos padrões e podem não aceitar alguns recursos do padrão. Mais informações sobre esses recursos podem ser encontradas nos manuais do usuário SQL dos respectivos produtos.

`java.sun.com/docs/books/tutorial` é uma excelente fonte para informações adicionais (e atualizadas) sobre JDBC e sobre Java em geral. Referências para livros sobre Java (incluindo JDBC) também estão disponíveis nesse URL. A API ODBC é descrita em Microsoft [1997] e Sanders [1998]. Melton e Eisenberg [2000] fornecem um guia para a SQLJ, JDBC e tecnologias relacionadas. Mais informações sobre ODBC, ADO e ADO.NET podem ser encontradas em `msdn.microsoft.com/data`.

CAPÍTULO 5

Outras linguagens relacionais

No Capítulo 2 apresentamos a álgebra relacional, que forma a base da amplamente usada linguagem de consulta SQL. A SQL foi discutida em grande detalhe nos Capítulos 3 e 4. Neste capítulo, primeiro estudaremos mais duas linguagens formais, o cálculo relacional de tupla e o cálculo relacional de domínio, que são linguagens de consulta declarativas baseadas na lógica matemática. Essas duas linguagens formais compõem a base para mais duas linguagens amigáveis, a QBE e a Datalog, que estudaremos mais adiante neste capítulo.

Diferente da SQL, a QBE é uma linguagem gráfica, em que as consultas *se parecem* com tabelas. A QBE e suas variantes são muito usadas nos sistemas de banco de dados em computadores pessoais. A Datalog possui uma sintaxe modelada baseada na linguagem Prolog. Embora não usada comercialmente no momento, a Datalog tem sido usada em vários sistemas de banco de dados de pesquisa.

Para a QBE e a Datalog, apresentamos construções e conceitos fundamentais em vez de um guia do usuário completo para essas linguagens. Tenha em mente que implementações individuais de uma linguagem podem diferir nos detalhes ou pode aceitar apenas um subconjunto da linguagem completa.

O cálculo relacional de tupla

Quando escrevemos uma expressão de álgebra relacional, fornecemos uma seqüência de procedimentos que gera a resposta para nossa consulta. O cálculo relacional de tupla, por outro lado, é uma linguagem de consulta **não procedural**. Ele descreve as informações desejadas sem fornecer um procedimento específico para obter essas informações.

Uma consulta no cálculo relacional de tupla é expressa como

$$\{t \mid P(t)\}$$

ou seja, ela é o conjunto de todas as tuplas t tais que o predicado P seja verdadeiro para t. Seguindo nossa notação anterior, usamos $t[A]$ para indicar o valor da tupla t no atributo A, e usamos $t \in r$ para indicar que a tupla t está na relação r.

Antes de uma definição formal do cálculo relacional de tupla, voltaremos para algumas das consultas para as quais escrevemos expressões de álgebra relacional na seção "Operações fundamentais da álgebra relacional" do Capítulo 2. Lembre-se de que as consultas são sobre o seguinte esquema:

agência(*nome_agência*, *cidade_agência*, *ativo*)
cliente (*nome_cliente*, *rua_cliente*, *cidade_cliente*)
empréstimo (*número_empréstimo*, *nome_agência*, *quantia*)
tomador (*nome_cliente*, *número_empréstimo*)
conta (*número_conta*, *nome_agência*, *saldo*)
depositante (*nome_cliente*, *número_conta*)

Consultas de exemplo

Encontre o *nome_agência*, o *número_empréstimo* e a *quantia* para empréstimos de mais de $1.200:

$$\{t \mid t \in \textit{empréstimo} \wedge t[\textit{quantia}] > 1200\}$$

Suponha que queremos apenas o atributo *número_empréstimo*, em vez de todos os atributos da relação *empréstimo*. Para escrever essa consulta no cálculo relacional de tupla, precisamos escrever uma expressão para uma relação no esquema (*número_empréstimo*). Precisamos das tuplas em (*número_empréstimo*) de modo que exista uma tupla em *empréstimo* com o atributo *quantia* > 1200. Para expressar

essa requisição, precisamos da construção "existe" da lógica matemática. A notação

$$\exists\, t \in r\, (Q(t))$$

significa "existe uma tupla t na relação r tal que o predicado $Q(t)$ seja verdadeiro."

Usando essa notação, podemos escrever a consulta "Encontre o número de empréstimo para cada empréstimo de uma quantia maior que $1.200" como

$$\{t \mid \exists\, s \in \textit{empréstimo}\ (t[\textit{número\_empréstimo}] = s[\textit{número\_empréstimo}] \wedge s[\textit{quantia}] > 1200)\}$$

Em português, lemos essa expressão como "O conjunto de todas as tuplas t tais que exista uma tupla s na relação *empréstimo* para a qual os valores de t e s para o atributo *número_empréstimo* sejam iguais, e o valor de s para o atributo *quantia* seja maior que $1.200."

A variável de tupla t é definida apenas no atributo *número_empréstimo*, já que esse é o único atributo tendo uma condição especificada por t. Portanto, o resultado é uma relação em (*número_empréstimo*).

Considere a consulta "Encontre os nomes de todos os clientes que possuem um empréstimo da agência Perryridge". Essa consulta é ligeiramente mais complexa do que as consultas anteriores, uma vez que ela envolve duas relações: *tomador* e *empréstimo*. Como veremos, entretanto, tudo o que ela exige é que tenhamos duas cláusulas "existe" em nossa expressão de cálculo relacional de tupla, conectadas por *and* ($\wedge$). Escrevemos a consulta desta maneira:

$$\{t \mid \exists\, s \in \textit{tomador}\ (t[\textit{nome\_cliente}] = s[\textit{nome\_cliente}] \wedge \exists\, u \in \textit{empréstimo}\ (u[\textit{número\_empréstimo}] = s[\textit{número\_empréstimo}] \wedge u[\textit{nome\_agência}] = \text{"Perryridge"}))\}$$

Em português, essa expressão é "O conjunto de todas as tuplas (*nome_cliente*) para as quais o cliente tem um empréstimo que esteja na agência Perryridge". A variável de tupla u assegura que o cliente é um tomador da agência Perryridge. A variável de tupla s é restrita para pertencer ao mesmo número de empréstimo de s. A Figura 5.1 mostra o resultado dessa consulta.

Para encontrar todos os clientes que possuem um empréstimo, uma conta ou as duas coisas no banco, usamos a operação união da álgebra relacional. No cálculo relacional de tupla, precisaremos de duas cláusulas "existe", conectadas por *or* ($\vee$):

$$\{t \mid \exists\, s \in \textit{tomador}\ (t[\textit{nome\_cliente}] = s[\textit{nome\_cliente}]) \vee \exists\, u \in \textit{depositante}\ (t[\textit{nome\_cliente}] = u[\textit{nome\_cliente}])\}$$

Essa expressão nos dá o conjunto de todas as tuplas *nome_cliente* para as quais pelo menos uma das seguintes condições se verifica:

- O *nome_cliente* aparece em alguma tupla da relação *tomador* como um tomador de empréstimo do banco.
- O *nome_cliente* aparece em alguma tupla da relação *depositante* como um depositante do banco.

Se algum cliente tiver tanto um empréstimo quanto uma conta no banco, esse cliente aparece apenas uma vez no resultado, pois a definição matemática de um conjunto não permite membros duplicados. O resultado dessa consulta apareceu anteriormente na Figura 2.11.

Se agora quisermos *apenas* os clientes que possuem uma conta e um empréstimo no banco, tudo o que precisamos fazer é mudar o *or* ($\vee$) para *and* ($\wedge$) na expressão anterior.

$$\{t \mid \exists\, s \in \textit{tomador}\ (t[\textit{nome\_cliente}] = s[\textit{nome\_cliente}]) \wedge \exists\, u \in \textit{depositante}\ (t[\textit{nome\_cliente}] = u[\textit{nome\_cliente}])\}$$

O resultado dessa consulta apareceu na Figura 2.19.

Agora, considere a consulta "Encontre todos os clientes que têm uma conta no banco mas não têm um empréstimo do banco". A expressão do cálculo relacional de tupla para essa consulta é semelhante às expressões que acabamos de ver, exceto pelo uso do símbolo *not* ($\neg$):

$$\{t \mid \exists\, u \in \textit{depositante}\ (t[\textit{nome\_cliente}] = u[\textit{nome\_cliente}]) \wedge \neg\, \exists\, s \in \textit{tomador}\ (t[\textit{nome\_cliente}] = s[\textit{nome\_cliente}])\}$$

Essa expressão de cálculo relacional de tupla usa a cláusula $\exists\, u \in \textit{depositante}\ (\ldots)$ para exigir que o cliente tenha

nome_cliente
Adams
Hayes

Figura 5.1 Nomes de todos os clientes que têm um empréstimo na agência Perryridge.

uma conta no banco, e usa a cláusula $\neg \exists s \in tomador\ (...)$ para eliminar os clientes que aparecem em alguma tupla da relação *tomador* como tendo um empréstimo do banco. O resultado dessa consulta apareceu na Figura 2.12.

A consulta que consideraremos a seguir usa implicação, indicada por $\Rightarrow$. A fórmula $P \Rightarrow Q$ significa "P implica Q"; ou seja, "se P é verdadeiro, então, Q precisa ser verdadeiro". Observe que $P \Rightarrow Q$ equivale logicamente a $\neg P \vee Q$. O uso da implicação em vez de *not* e *or* normalmente sugere uma interpretação mais intuitiva de uma consulta em português.

Considere a consulta que usamos na seção "A operação divisão" do Capítulo 2 para ilustrar a operação divisão: "Encontre todos os clientes que têm uma conta em todas as agências localizadas em Brooklyn". Para escrever essa consulta no cálculo relacional de tupla, introduzimos a construção "para todo", indicada por $\forall$. A notação

$$\forall\ t \in r\ (Q(t))$$

significa "Q é verdadeiro para todas as tuplas t na relação r".

Escrevemos a expressão para nossa consulta desta forma:

$$\{t \mid \exists\ r \in cliente\ (r[nome\_cliente] = t[nome\_cliente]) \wedge (\forall\ u \in agência\ (u[cidade\_agência] = \text{"Brooklyn"} \Rightarrow \exists\ s \in depositante\ (t[nome\_cliente] = s[nome\_cliente] \wedge \exists\ w \in conta\ (w[numero\_conta] = s[numero\_conta] \wedge w[nome\_agência] = u[nome\_agência])))))\}$$

Em português, interpretamos essa expressão como "O conjunto de todos os clientes (ou seja, tuplas t (*nome_cliente*)), tal que, para todas as tuplas u na relação *agência*, se o valor de u no atributo *cidade_agência* é Brooklyn, então, o cliente tem uma conta na agência cujo nome aparece no atributo *nome_agência* de u".

Repare que existe uma sutileza nessa consulta: se não houver agência alguma em Brooklyn, todos os nomes de cliente satisfazem a condição. A primeira linha da expressão de consulta é vital nesse caso – sem a condição

$$\exists\ r \in cliente\ (r[nome\_cliente] = t[nome\_cliente])$$

se não houver agência alguma em Brooklyn, qualquer valor de t (inclusive valores que não são nomes de cliente na relação *cliente*) se qualificará.

Definição formal

Agora estamos prontos para uma definição formal. Uma expressão de cálculo relacional de tupla é da forma

$$\{t \mid P(t)\}$$

onde P é uma *fórmula*. Diversas variáveis de tupla podem aparecer em uma fórmula. Uma variável de tupla é considerada como uma *variável livre*, a menos que seja quantificada por um $\exists$ ou um $\forall$. Assim, em

$$t \in empréstimo \wedge \exists\ s \in cliente\ (t[nome\_agência] = s[nome\_agência])$$

t é uma variável livre. A variável de tupla é chamada de uma variável *limitada*.

Uma fórmula de cálculo relacional de tupla é constituída de *átomos*. Um átomo tem uma das formas a seguir:

- $s \in r$, onde s é uma variável de tupla e r é uma relação (não permitimos o uso do operador $\notin$)
- $s[x]\ \Theta\ u[y]$, onde s e u são variáveis de tupla, x é um atributo no qual s está definido, y é um atributo no qual u está definido e Θ é um operador de comparação ($<, \leq, =, \neq, >, \geq$); exigimos que os atributos x e y tenham domínios cujos membros podem ser comparado por Θ
- $s[x]\ \Theta\ c$, onde s é uma variável de tupla, x é um atributo no qual s está definido, Θ é um operador de comparação e c é uma constante no domínio do atributo x

Construímos fórmulas a partir de átomos usando as seguintes regras:

- Um átomo é uma fórmula.
- Se P_1 é uma fórmula, então, $\neg P_1$ e (P_1) também são fórmulas.
- Se P_1 e P_2 são fórmulas, então, $P_1 \vee P_2$, $P_1 \wedge P_2$ e $P_1 \Rightarrow P_2$ também são fórmulas.
- Se $P_1(s)$ é uma fórmula contendo uma variável de tupla livre s, e r é uma relação, então,

$$\exists\ s \in r\ (P_1(s)) \text{ e } \forall\ s \in r\ (P_1(s))$$

também são fórmulas.

Assim como na álgebra relacional, também podemos escrever expressões equivalentes que não sejam idênticas na aparência. No cálculo relacional de tupla, essas equivalências incluem as três seguintes regras:

1. $P_1 \wedge P_2$ é equivalente a $\neg(\neg(P_1) \vee \neg(P_2))$.
2. $\forall\ t \in r\ (P_1(t))$ é equivalente a $\neg \exists\ t \in r\ (\neg P_1(t))$.
3. $P_1 \Rightarrow P_2$ é equivalente a $\neg(P_1) \vee P_2$.

Segurança de expressões

Existe um último aspecto a ser considerado. Uma expressão de cálculo relacional de tupla pode gerar uma relação infinita. Suponha que queiramos escrever a expressão

$$\{t \mid \neg(t \in empréstimo)\}$$

Existe infinitamente muitas tuplas que não estão em *empréstimo*. A maioria dessas tuplas contém valores que nem mesmo aparecem no banco de dados! Claramente, não desejamos permitir essas expressões.

Para ajudar a definir uma restrição do cálculo relacional de tupla, introduzimos o conceito de **domínio** de uma fórmula relacional de tupla, *P*. Intuitivamente, o domínio de *P*, indicado por *dom*(*P*), é o conjunto de todos os valores referenciados por *P*. Eles incluem valores mencionados na própria *P*. Portanto, o domínio de *P* é o conjunto de todos os valores que aparecem explicitamente em *P* ou que aparecem em uma ou mais relações cujos nomes aparecem em *P*. Por exemplo, *dom*($t \in empréstimo \wedge t[quantia] > 1200$) é o conjunto contendo 1200 bem como o conjunto de todos os valores que aparecem em *empréstimo*. Além disso, *dom*($\neg(t \in empréstimo)$) é o conjunto de todos os valores aparecendo em *empréstimo*, já que a relação *empréstimo* é mencionada na expressão.

Dizemos que uma expressão $\{t \mid P(t)\}$ é *segura* se todos os valores que aparecem no resultado são valores de *dom*(*P*). A expressão $\{t \mid \neg(t \in empréstimo)\}$ não é segura. Note que *dom*($\neg(t \in empréstimo)$) é o conjunto de todos os valores aparecendo em *empréstimo*. Entretanto, é possível ter uma tupla *t* que não estejam em *empréstimo* que contenha valores que não aparecem em *empréstimo*. Os outros exemplos de expressões de cálculo relacional de tupla que escrevemos nesta seção são seguros.

Poder expressivo das linguagens

No que se refere ao poder expressivo, o cálculo relacional de tupla restrito às expressões seguras é equivalente à álgebra relacional básica (com os operadores $\cup$, $-$, $\times$, σ e ρ, mas sem os operadores relacionais estendidos, como projeção generalizada $\mathcal{G}$ e as operações de junção externa). Portanto, para cada expressão de álgebra relacional usando apenas as operações básicas, existe uma expressão equivalente no cálculo relacional de tupla; e para cada expressão de cálculo relacional de tupla, existe uma expressão de álgebra relacional equivalente. Não demonstraremos essa afirmação aqui; as notas bibliográficas contêm as referências. Algumas partes da demonstração são incluídas nos exercícios. Observe que o cálculo relacional de tupla não possui qualquer equivalente da operação de agregação, mas ele pode ser estendido para dar suporte à agregação. É simples estender o cálculo relacional de tupla para manipular expressões aritméticas.

O cálculo relacional de domínio

Uma segunda forma de cálculo relacional, chamada **cálculo relacional de domínio**, usa variáveis de *domínio* que assumem valores de um domínio de atributos, em vez de valores para uma tupla inteira. O cálculo relacional de domínio, no entanto, está intimamente relacionado ao cálculo relacional de tupla.

O cálculo relacional de domínio serve como a base teórica da amplamente usada linguagem QBE, assim como a álgebra relacional serve como a base para a linguagem SQL.

Definição formal

Uma expressão no cálculo relacional de domínio tem a forma

$$\{< x_1, x_2, ..., x_n > \mid P(x_1, x_2, ..., x_n)\}$$

onde $x_1, x_2, ..., x_n$ representam variáveis de domínio. *P* representa uma fórmula composta de átomos, como foi o caso no cálculo relacional de tupla. Um átomo no cálculo relacional de domínio tem uma das seguintes formas:

- $< x_1, x_2, ..., x_n > \in r$, onde *r* é uma relação nos atributos *n* e $x_1, x_2, ..., x_n$ são variáveis de domínio ou constantes de domínio.
- $x \Theta y$, onde *x* e *y* são variáveis de domínio e Θ é um operador de comparação ($<, \leq, =, \neq, >, \geq$); exigimos que os atributos *x* e *y* tenham domínios que podem ser comparados por Θ.
- $x \Theta c$, onde *x* é uma variável de domínio, Θ é um operador de comparação e *c* é uma constante no domínio do atributo para o qual *x* é uma variável de domínio.

Construímos fórmulas a partir de átomos usando as seguintes regras:

- Um átomo é uma fórmula.
- Se P_1 é uma fórmula, então, $\neg P_1$ e (P_1) também são fórmulas.
- Se P_1 e P_2 são fórmulas, então, $P_1 \vee P_2$, $P_1 \wedge P_2$ e $P_1 \Rightarrow P_2$ também são fórmulas.
- Se $P_1(x)$ é uma fórmula em *x*, onde *x* é uma variável de domínio livre, então,

$$\exists x\ (P_1(x)) \text{ e } \forall x\ (P_1(x))$$

também são fórmulas.

Como uma abreviatura notacional, escrevemos $\exists a, b, c\ (P(a,b,c))$ para $\exists a\ (\exists b\ (\exists c\ (P(a,b,c))))$.

Consultas de exemplo

Agora fornecemos consultas de cálculo relacional de domínio para os exemplos que consideramos anteriormente. Repare a semelhança entre essas expressões e as expressões de cálculo relacional de tupla correspondentes.

- Encontre o número de empréstimo, o nome da agência e a quantia para empréstimos de mais de $1.200:

$$\{< l,b,a > \mid < l,b,a > \in \textit{empréstimo} \wedge a > 1.200\}$$

- Encontre todos os números de empréstimo para empréstimos com uma quantia maior que $1200:

$$\{< l > \mid \exists\, b,a\ (< l,b,a > \in \textit{empréstimo} \wedge a > 1200)\}$$

Embora a segunda consulta pareça semelhante à que escrevemos para o cálculo relacional de tupla, existe uma diferença importante. No cálculo de tupla, quando escrevemos $\exists\, s$ para alguma variável de tupla s, a vinculamos imediatamente a uma relação escrevendo $\exists\, s \in r$. Entretanto, quando escrevemos $\exists\, b$ no cálculo de domínio, b se refere não a uma tupla, mas a um valor de domínio. Portanto, o domínio da variável b é irrestrito até que a subfórmula $< l,b,a > \in \textit{empréstimo}$ restrinja b aos nomes de agência que aparecem na relação *empréstimo*.

Agora, forneceremos vários exemplos de consultas no cálculo relacional de domínio.

- Encontre os nomes de todos os clientes que possuem um empréstimo na agência Perryridge e encontre a quantia do empréstimo:

$$\{< c,a > \mid \exists\, l\ (< c,l > \in \textit{tomador} \wedge \exists\, b\ (< l,b,a > \in \textit{empréstimo} \wedge b = \text{"Perryridge"}))\}$$

- Encontre os nomes de todos os clientes que possuem um empréstimo, uma conta ou ambos na agência Perryridge:

$$\{< c > \mid \exists\, l\ (< c,l > \in \textit{tomador} \wedge \exists\, b,a\ (< l,b,a > \in \textit{empréstimo} \wedge b = \text{"Perryridge"})) \vee \exists\, a\ (< c,a > \in \textit{depositante} \wedge \exists\, b,n\ (< a,b,n > \in \textit{conta} \wedge b = \text{"Perryridge"}))\}$$

- Encontre os nomes de todos os clientes que possuem uma conta em todas as agências localizadas em Brooklyn:

$$\{< c > \mid \exists\, s,t\ (< c,s,t > \in \textit{cliente}) \wedge \forall\, x,y,z\ (< x,y,z > \in \textit{agência} \wedge y = \text{"Brooklyn"} \Rightarrow \exists\, a,b\ (< a,x,b > \in \textit{conta} \wedge < c,a > \in \textit{depositante}))\}$$

Em português, interpretamos essa expressão como "O conjunto de todas as tuplas (*nome_cliente*) c tal que, para todas as tuplas x,y,z (*nome_agência, cidade_agência, ativo*), se a cidade da agência for Brooklyn, então, o seguinte é verdadeiro:

- □ Existe uma tupla na relação *conta* com o número de conta a e o nome de agência x.
- □ Existe uma tupla na relação *depositante* com o cliente c e o número de conta a."

Segurança de expressões

Observamos que, no cálculo relacional de tupla (primeira seção deste capítulo), é possível escrever expressões que podem gerar uma relação infinita. Isso nos levou a definir *segurança* para expressões de cálculo relacional de tupla. Uma situação semelhante ocorre para o cálculo relacional de domínio. Uma expressão como

$$\{< l,b,a > \mid \neg(< l,b,a > \in \textit{empréstimo})\}$$

não é segura, pois ela permite que valores no resultado não estejam no domínio da expressão.

Para o cálculo relacional de domínio, precisamos nos preocupar com a forma das fórmulas dentro das cláusulas "existe" e "para todo". Considere a expressão

$$\{< x > \mid \exists\, y\ (< x,y > \in r) \wedge \exists\, z\ (\neg(< x,z > \in r) \wedge P(x,z))\}$$

onde P é alguma fórmula envolvendo x e z. Podemos testar a primeira parte da fórmula, $\exists\, y\ (< x,y > \in r)$, considerando apenas os valores em r. Entretanto, para testar a segunda parte da fórmula, $\exists\, z\ (\neg(< x,z > \in r) \wedge P(x,z))$, precisamos considerar valores para z que não aparecem em r. Como todas as relações são finitas, um número infinito de valores não aparece em r. Portanto, geralmente não é possível testar a segunda parte da fórmula sem considerar um número infinito de possíveis valores para z. Em vez disso, acrescentamos restrições para proibir expressões como a anterior.

No cálculo relacional de tupla, restringimos qualquer variável existencialmente qualificada para variar sobre uma relação específica. Como não fizemos isso no cálculo de domínio, acrescentamos regras à definição de segurança para lidar com casos como nosso exemplo. Dizemos que uma expressão

$$\{< x_1, x_2, \ldots, x_n > \mid P\ (x_1, x_2, \ldots, x_n)\}$$

é segura se todos os seguintes se verificam:

1. Todos os valores que aparecem em tuplas da expressão são valores de $dom(P)$.
2. Para toda subfórmula "existe" da forma $\exists\, x\ (P_1(x))$, a subfórmula é verdadeira se e somente se existir um valor x em $dom(P_1)$ tal que $P_1(x)$ seja verdadeiro.
3. Para toda subfórmula "para todo" da forma $\forall\, x\ (P_1(x))$, a subfórmula é verdadeira se e somente se $P_1(x)$ for verdadeiro para todos os valores x de $dom(P_1)$.

A finalidade das regras adicionais é garantir que possamos testar subfórmulas "para todo" e "existe" sem precisar testar infinitamente muitas possibilidades. Considere a segunda regra na definição de segurança. Para $\exists x\ (P_1(x))$ ser verdadeiro, precisamos encontrar apenas um x para o qual $P_1(x)$ é verdadeiro. Em geral, haveria infinitamente muitos valores para testar. Entretanto, se a expressão é segura, sabemos que podemos restringir nossa atenção aos valores de $dom(P_1)$. Essa restrição reduz para um número finito de tuplas que precisamos considerar.

A situação para subfórmulas da forma $\forall x\ (P_1(x))$ é semelhante. Para afirmar que $\forall x\ (P_1(x))$ é verdadeiro, normalmente precisamos testar todos os valores possíveis e, portanto, temos de examinar infinitamente muitos valores. Como antes, se sabemos que a expressão é segura, isso é suficiente para testarmos $P_1(x)$ para os valores tomados de $dom(P_1)$.

Todas as expressões de cálculo relacional de domínio que escrevemos nas consultas de exemplo desta seção são seguras.

Poder expressivo das linguagens

Quando o cálculo relacional de domínio é restrito a expressões seguras, ele equivale em poder expressivo ao cálculo relacional de tupla restrito a expressões seguras. Como observamos anteriormente que o cálculo relacional de tupla restrito é equivalente à álgebra relacional, os três itens a seguir são equivalentes:

- A álgebra relacional básica (sem as operações de álgebra relacional estendidas)
- O cálculo relacional de tupla restrito a expressões seguras
- O cálculo relacional de domínio restrito a expressões seguras

Convém notar que o cálculo relacional de domínio também não possui qualquer equivalente da operação de agregação, mas pode ser estendido para aceitar agregação; além disso, é muito simples estendê-lo para manipular expressões aritméticas.

Query-by-Example

Query-by-Example (QBE) é o nome de uma linguagem de manipulação de dados e também de um sistema de banco de dados antigo que incluía essa linguagem.

A linguagem de manipulação de dados QBE possui dois recursos característicos:

1. Diferente da maioria das linguagens de consulta e das linguagens de programação, a QBE possui uma **sintaxe bidimensional**. As consultas se *parecem* com tabelas. Uma consulta em uma linguagem unidimensional (por exemplo, SQL) *pode* ser escrita em uma linha (possivelmente longa). Uma linguagem bidimensional *exige* duas dimensões para sua expressão. (Existe uma versão unidimensional da QBE, mas não iremos considerá-la em nossa discussão.)
2. As consultas QBE são expressas "por exemplo". Em vez de fornecer um procedimento para obter a resposta desejada, o usuário fornece um exemplo do que é desejado. O sistema generaliza esse exemplo para calcular a resposta à consulta.

Apesar desses recursos incomuns, existe uma correspondência próxima entre a QBE e o cálculo relacional de domínio.

Existem dois tipos de QBE: a versão original baseada em texto e uma versão gráfica desenvolvida mais tarde que é aceita pelo sistema de banco de dados Microsoft Access. Nesta seção, fornecemos um breve resumo dos recursos de manipulação de dados das duas versões da QBE. Primeiro abordaremos os recursos da QBE baseada em texto que correspondem à cláusula SQL select-from-where sem agregação ou atualizações. Veja nas notas bibliográficas as referências de onde você pode obter mais informações sobre como a versão baseada em texto da QBE manipula a classificação da saída, a agregação e a atualização. Mais adiante, na seção "QBE no Microsoft Access", mais adiante neste capítulo, veremos brevemente os recursos da versão gráfica da QBE.

Tabelas de estrutura

Expressamos consultas na QBE por **tabelas de estrutura**. Essas tabelas mostram o esquema de relação, como na Figura 5.2. Em vez de encher a tela com todas as estruturas, o usuário seleciona as estruturas necessárias para uma determinada consulta e preenche as estruturas com linhas de exemplo. Uma linha de exemplo consiste em constantes e *elementos de exemplo*, que são variáveis de domínio. Para evitar confusão entre as duas, a QBE usa um caractere de sublinhado (_) antes das variáveis de domínio, como em _x, e deixa as constantes aparecerem sem qualquer qualificação. Essa convenção está em desacordo com a maioria das outras linguagens, em que as constantes aparecem entre aspas e as variáveis aparecem sem qualquer qualificação.

Consultas em uma relação

Voltando ao nosso exemplo de banco, para encontrar todos os números de empréstimo na agência Perryridge, montamos a estrutura para a relação *empréstimo* e a preenchemos da seguinte maneira:

agência	nome_agência	cidade_agência	ativo

cliente	nome_cliente	rua_cliente	cidade_cliente

empréstimo	número_empréstimo	nome_agência	quantia

tomador	nome_cliente	número_empréstimo

quantia	número_conta	nome_agência	saldo

depositante	nome_cliente	número_conta

Figura 5.2 Tabelas de estrutura QBE para o exemplo de banco.

empréstimo	número_empréstimo	nome_agência	quantia
	P._x	Perryridge	

Essa consulta diz ao sistema para procurar tuplas na relação *empréstimo* que têm "Perryridge" como o valor para o atributo *nome_agência*. Para cada uma dessas tuplas, o sistema atribui o valor do atributo *número_empréstimo* à variável *x*. Ele "imprime" (na verdade, exibe) o valor da variável *x*, pois o comando P. aparece na coluna *número_empréstimo* adjacente à variável *x*. Observe que esse resultado é semelhante ao que seria feito para responder à consulta de cálculo relacional de domínio

$$\{\langle x \rangle \mid \exists\, b, a(\langle x,b,a \rangle \in \textit{empréstimo} \wedge b = \text{"Perryridge"})\}$$

A QBE considera que uma posição vazia em uma linha contém uma variável única. Desse modo, se uma variável não aparece mais de uma vez em uma consulta, ela pode ser omitida. Assim, nossa consulta anterior poderia ser reescrita como

empréstimo	número_empréstimo	nome_agência	quantia
	P.	Perryridge	

A QBE (diferente da SQL) realiza a eliminação de duplicatas automaticamente. Para suprimir a eliminação de duplicatas, inserimos o comando ALL. após o comando P., desta maneira:

empréstimo	número_empréstimo	nome_agência	quantia
	P.ALL.	Perryridge	

Para exibir a relação *empréstimo* inteira, podemos criar uma única linha contendo P. em cada campo. Como alternativa, podemos usar uma notação abreviada colocando um único P. na coluna que tem o nome da relação como cabeçalho:

empréstimo	número_empréstimo	nome_agência	quantia
P.			

A QBE permite consultas que chamam comparações aritméticas (por exemplo, >), em vez de comparações de igualdade, como em "Encontre os números de empréstimo de todos os empréstimos com quantia de mais de $700":

empréstimo	número_ empréstimo	nome_agência	quantia
	P.		>700

As comparações podem chamar apenas uma expressão aritmética no lado direito da operação de comparação (por exemplo, > (_x + _y – 20)). A expressão pode incluir tanto variáveis quanto constantes. O espaço no lado esquerdo da operação de comparação precisa estar vazio. As operações aritméticas aceitas pela QBE são =, <, ≤, >, ≥ e ¬.

Observe que a exigência de que o lado esquerdo esteja vazio implica que não podemos comparar duas variáveis nomeadas distintas. Lidaremos com essa dificuldade em breve.

Como outro exemplo, considere a consulta "Encontre os nomes de todas as agências que não estão localizadas em Brooklyn". Essa consulta pode ser escrita da seguinte forma:

agência	nome_agência	cidade_agência	ativo
	P.	¬ Brooklyn	

A principal finalidade das variáveis na QBE é obrigar valores de certas tuplas a terem o mesmo valor em certos atributos. Considere a consulta "Encontre os números de empréstimo de todos os empréstimos feitos conjuntamente para Smith e Jones":

tomador	nome_cliente	número_empréstimo
	Smith	P._x
	Jones	_x

Para executá-la, o sistema encontra todas as tuplas em *tomador* que concordam no atributo *número_empréstimo*, onde o valor para o atributo *nome_cliente* é "Smith" para uma tupla e "Jones" para a outra. O sistema, então, exibe o valor do atributo *número_empréstimo*.

No cálculo relacional de domínio, a consulta seria escrita como

$$\{\langle l\rangle \mid \exists\, x\, (\langle x,l\rangle \in tomador \wedge x = \text{"Smith"}) \wedge \exists\, x\, (\langle x,l\rangle \in tomador \wedge x = \text{"Jones"})\}$$

Ainda outro exemplo, considere a consulta "Encontre todos os clientes que moram na mesma cidade de Jones":

cliente	nome_cliente	rua_cliente	cidade_cliente
	P._x		_y
	Jones		_y

Consultas em várias relações

A QBE permite que as consultas abranjam várias relações diferentes (analogamente ao produto cartesiano ou à junção natural na álgebra relacional). As conexões entre as várias relações são obtidas por meio de variáveis que obrigam certas tuplas a terem o mesmo valor em certos atributos. Como uma ilustração, suponha que queremos encontrar os nomes de todos os clientes que têm um empréstimo na agência Perryridge. Essa consulta pode ser escrita como

empréstimo	número_ empréstimo	nome_agência	quantia
	_x	Perryridge	

Tomador	nome_cliente	número_empréstimo
	P._y	_x

Para avaliar a consulta anterior, o sistema encontra tuplas em *empréstimo* com "Perryridge" como o valor para o atributo *nome_agência*. Para cada uma dessas tuplas, o sistema encontra tuplas em *tomador* com o mesmo valor para o atributo *número_empréstimo* que a tupla *empréstimo*. Ele exibe os valores para o atributo *nome_cliente*.

Podemos usar uma técnica semelhante para escrever a consulta "Encontre os nomes de todos os clientes que têm uma conta e um empréstimo no banco":

Depositante	nome_cliente	número_conta
	P._x	

Tomador	nome_cliente	número_empréstimo
	_x	

Agora, considere a consulta "Encontre os nomes de todos os clientes que têm uma conta no banco, mas que não têm um empréstimo do banco". Expressamos consultas que implicam negação na QBE colocando um sinal not (¬) sob o nome da relação e ao lado de uma linha de exemplo:

Depositante	nome_cliente	número_conta
	P._x	

Tomador	nome_cliente	número_empréstimo
¬	_x	

Compare essa consulta com a consulta anterior "Encontre os nomes de todos os clientes que têm uma conta e um empréstimo no banco". A única diferença é o sinal ¬ aparecendo ao lado da linha de exemplo na estrutura *tomador*. Essa diferença, contudo, tem um importante efeito no pro-

cessamento da consulta. A QBE encontra todos os valores x para os quais

1. Existe uma tupla na relação *depositante* cujo *nome_cliente* é a variável de domínio x.
2. Não existe qualquer tupla na relação *tomador* cujo *nome_cliente* é o mesmo que na variável de domínio x.

O ¬ pode ser lido como "não existe".

O fato de termos colocado o ¬ sob o nome da relação, e não sob um nome de atributo, é importante. Um ¬ sob um nome de atributo é abreviatura de ≠. Portanto, para encontrar todos os clientes que têm pelo menos duas contas, escrevemos

depositante	nome_cliente	número_conta
	P._x	_y
	_x	¬ _y

Em português, a consulta anterior é lida como "Exiba todos os valores *nome_cliente* que aparecem em pelo menos duas tuplas, com a segunda tupla tendo um *número_conta* diferente do primeiro".

A caixa de condição

Às vezes, é inconveniente ou impossível expressar todas as restrições nas variáveis de domínio dentro das tabelas de estrutura. Para contornar essa dificuldade, a QBE inclui um recurso de **caixa de condição**, que permite a expressão de restrições gerais sobre qualquer uma das variáveis de domínio. A QBE permite que expressões lógicas apareçam em uma caixa de condição. Os operadores lógicos são as palavras **and** e **or**, ou os símbolos "&" e "|".

Por exemplo, a consulta "Encontre os números de empréstimo de todos os empréstimos feitos para Smith, para Jones (ou para ambos conjuntamente)" pode ser escrita como

tomador	nome_cliente	número_empréstimo
	_n	P._x

Condições
_n = Smith or _n = Jones

É possível expressar essa consulta sem usar uma caixa de condição, usando P. em várias linhas. Entretanto, as consultas com P. em várias linhas algumas vezes são difíceis de entender e devem ser evitadas, se possível.

Ainda como outro exemplo, suponha que modifiquemos a consulta final na seção anterior para "Encontre todos os clientes que não têm o nome 'Jones' e que têm pelo menos duas contas". Queremos incluir uma restrição "$x \neq$ Jones" nessa consulta. Isso é feito montando a caixa de condição e inserindo a restrição "$x \neg =$ Jones":

Condições
x ¬ = Jones

Voltando para outro exemplo, para encontrar todos os números de conta com um saldo entre $1.300 e $1.500, escrevemos

conta	número_conta	nome_agência	saldo
	P.		_x

Condições
_x ≥ 1300
_x ≤ 1500

Como outro exemplo, considere a consulta "Encontre todas as agências que possuem ativos maiores que os de pelo menos uma agência localizada em Brooklyn". Essa consulta pode ser escrita como

agência	nome_agência	cidade_agência	ativo
	P._x		_y
		Brooklin	_z

Condições
_y > _z

A QBE permite que expressões aritméticas complexas apareçam em uma caixa de condição. Podemos escrever a consulta "Encontre todas as agências que possuem ativos que sejam pelo menos o dobro do ativo de uma das agências localizadas em Brooklyn" quase da mesma forma que escrevemos na consulta anterior, modificando a caixa de condição para

Condições
_y ≥ 2 * _z

Para encontrar o número de conta das contas com um saldo entre $1.300 e $2.000, mas não exatamente $1.500, escrevemos

conta	número_conta	nome_agência	saldo
	P.		_x

condições
_x = (≥ 1300 and ≤ 2000 and ¬ 1500)

A QBE usa a construção **or** de uma maneira não convencional para permitir comparação com um conjunto de valores constantes. Para encontrar todas as agências que estão localizadas em Brooklyn ou Queens, escrevemos

agência	*nome_agência*	*cidade_agência*	*ativos*
P.		_x	

condições
_x = (Brooklyn **or** Queens)

A relação resultado

As consultas que escrevemos até agora possuem uma característica em comum: os resultados a serem exibidos aparecem em um único esquema de relação. Se o resultado de uma consulta inclui atributos de vários esquemas de relação, precisamos de um mecanismo para exibir o resultado desejado em uma única tabela. Para essa finalidade, podemos declarar uma relação *resultado* temporária que inclui todos os atributos do resultado da consulta. Imprimimos o resultado desejado incluindo o comando P. apenas na tabela de estrutura do *resultado*.

Como ilustração, considere a consulta "Encontre o *nome_cliente*, o *número_conta* e o *saldo* para todas as contas na agência Perryridge". Em álgebra relacional, construiríamos essa consulta da seguinte maneira:

1. Com junção de *depositante* e *conta*.
2. Com projeção de *nome_cliente*, *número_conta* e *saldo*.

Para construir a mesma consulta na QBE, fazemos o seguinte:

1. Criamos uma tabela de estrutura, chamada *resultado*, com atributos *nome_cliente*, *número_conta* e *saldo*. O nome da tabela de estrutura recém-criada (ou seja, *resultado*) precisa ser diferente de qualquer um dos nomes de relação de banco de dados anteriormente existentes.
2. Escrevemos a consulta.

A consulta resultante é

conta	*número_conta*	*nome_agência*	*saldo*
	_y	Perryridge	_z

depositante	*Nome_cliente*	*número_conta*
	_x	_y

resultado	*nome_cliente*	*número_conta*	*saldo*
P.	_x	_y	_z

QBE no Microsoft Access

Nesta seção, examinaremos a versão da QBE aceita pelo Microsoft Access. Enquanto a QBE original foi projetada para um ambiente de exibição baseado em texto, a Access QBE é projetada para um ambiente de exibição gráfico e apropriadamente chamado de **Graphical Query-By-Example** (GQBE).

A Figura 5.3 mostra uma consulta de exemplo da GQBE. A consulta pode ser descrita em português como "Encontre o *nome_cliente*, o *número_conta* e o *saldo* para todas as contas na agência Perryridge". A seção anterior mostrou como ela é executada em QBE.

Uma diferença sutil na versão GQBE é que os atributos de uma tabela são escritos um abaixo do outro, em vez de

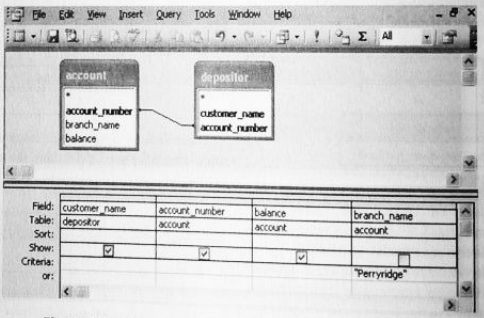


Figura 5.3 Uma consulta de exemplo na Microsoft Access QBE.

horizontalmente. Uma diferença mais significativa é que a versão gráfica da QBE usa uma linha vinculando atributos de duas tabelas, em vez de uma variável compartilhada, para especificar uma condição de junção.

Um recurso interessante da Access QBE é que os vínculos entre as tabelas são criados automaticamente, na base do nome de atributo. No exemplo da Figura 5.3, as duas tabelas, *conta* e *depositante*, foram incluídas na consulta. O atributo *número_conta* é compartilhado entre as duas tabelas selecionadas, e o sistema insere automaticamente um vínculo entre as duas tabelas. Em outras palavras, uma condição de junção natural é imposta como padrão entre as tabelas; o vínculo pode ser excluído se ele não for desejado. O vínculo também pode ser especificado para indicar uma junção externa natural, em vez de uma junção natural.

Outra pequena diferença na Access QBE é que ela especifica atributos para serem impressos em uma caixa separada, chamada **grade de projeto**, em vez de usar um P. na tabela. Ela também especifica seleções em valores de atributo na grade de projeto.

As consultas envolvendo group by e agregação podem ser criadas no Access, como mostra a Figura 5.4. A consulta nessa figura encontra o nome, a rua e a cidade de todos os clientes que possuem mais de uma conta no banco. Os atributos "group by", bem como as funções aggregate, são notados na grade de projeto.

Repare que quando uma condição aparece em uma coluna da grade de projeto com a linha "Total" definida em "aggregate", a condição é aplicada no valor agregado; por exemplo, na Figura 5.4, a seleção "> 1" na coluna *número_conta* é aplicada no resultado da aggregate "Count". Essas seleções correspondem às seleções em uma cláusula **having** da SQL.

As condições de seleção podem ser aplicadas em colunas da grade de projeto que não estão "agrupadas por" nem agregadas; esses atributos precisam ser marcados como "Where" na linha "Total". Essas seleções "Where" são aplicadas antes da agregação e correspondem às seleções em uma cláusula **where** da SQL. Entretanto, essas colunas não podem ser impressas (marcadas como "Show"). Apenas as colunas em que a linha "Total" especifica "group by" ou uma função aggregate podem ser impressas.

As consultas são criadas via uma interface gráfica com o usuário, primeiramente selecionando tabelas. Os atributos podem, então, ser incluídos na grade de projeto arrastando-os das tabelas e soltando-os na grade. Em seguida, as condições de seleção, o agrupamento e a agregação podem ser especificados nos atributos na grade de projeto. A Access QBE aceita vários outros recursos também, incluindo consultas para modificar o banco de dados por inserção, exclusão ou atualização.

Datalog

A Datalog é uma linguagem de consulta não procedural baseada na linguagem de programação lógica Prolog. Como no cálculo relacional, um usuário descreve as informações desejadas sem fornecer um procedimento específico para obter essas informações. A sintaxe da Datalog se assemelha à da Prolog. Entretanto, como o significado dos programas Datalog é definido de uma maneira puramente declarativa, diferente das semânticas mais procedurais da Prolog, a Datalog simplifica a escrita de consultas simples e facilita a otimização das consultas.

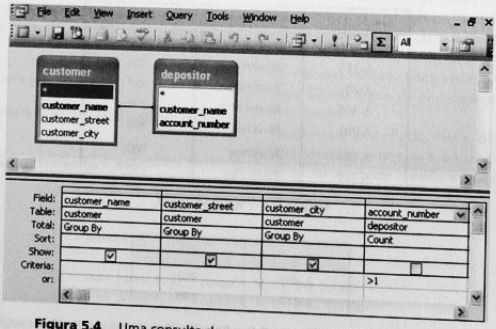


Figura 5.4 Uma consulta de agregação na Microsoft Access QBE.

Estrutura básica

Um programa Datalog consiste em um conjunto de regras. Antes de apresentarmos uma definição formal das regras Datalog e o seu significado formal, consideraremos alguns exemplos. Considere uma regra da Datalog para definir uma relação de view *v1* contendo números de conta e saldos para contas na agência Perryridge com saldo acima de $700:

$$v1(A, B) :- conta(A, \text{"Perryridge"}, B), B > 700$$

As regras Datalog definem views; a regra anterior usa a relação *conta* e define a relação de view *v1*. O símbolo :– é lido como "se", e a vírgula separando "*conta* (*A*, "Perryridge", *B*)" de "*B* > 700" é lida como "e". Intuitivamente, a regra é entendida como:

para todo *A*, *B*
if (*A*, "Perryridge", *B*) ∈ *conta* **and** *B* > 700
then (*A*, *B*) ∈ *v1*

Suponha que a relação *conta* é como mostra a Figura 5.5. Então, a relação view *v1* contém as tuplas na Figura 5.6.

Para recuperar o saldo da conta de número A-217 na relação de view *v1*, podemos escrever a seguinte consulta:

$$? v1 (\text{"A-217"}, B)$$

A resposta para a consulta é

(A-217, 750)

Para obter o número de conta e o saldo de todas as contas na relação *v1* em que o saldo é maior que $800, podemos escrever

$$? v1 (A, B), B > 800$$

A resposta para essa consulta é

(A-201, 900)

Em geral, precisamos de mais de uma regra para definir uma relação de view. Cada regra define um conjunto de tuplas que a relação de view precisa conter. O conjunto de tuplas na relação de view é definido como a união de todos esses conjuntos de tuplas. O programa Datalog a seguir especifica as taxas de juros para contas:

$$taxa\_juros(A, 5) :- conta(A, N, B), B < 10000$$
$$taxa\_juros(A, 6) :- conta(A, N, B), B >= 10000$$

O programa possui duas regras definindo uma relação de view *taxa_juros*, cujos atributos são o número de conta e a taxa de juros. As regras dizem que, se o saldo é menor do que $10.000, então, a taxa de juros é de 5%; e se o saldo é maior ou igual a $10.000, a taxa de juros é de 6%.

As regras Datalog também podem usar negação. As seguintes regras definem uma relação de view *c* que contém os nomes de todos os clientes que possuem um depósito mas não têm um empréstimo no banco:

$$c(N) :- depositante(N, A), \textbf{not } é\_tomador(N)$$
$$é\_tomador(N) :- tomador(N, L)$$

número_conta	*Nome_agência*	*saldo*
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Perryridge	900
A-222	Redwood	700
A-217	Perryridge	750

Figura 5.5 Relação *conta*.

número_conta	*saldo*
A-201	900
A-217	750

Figura 5.6 Relação *v1*.

As implementações da Prolog e a maioria das implementações da Datalog reconhecem atributos de uma relação por posição e omitem os nomes de atributo. Assim, as regras Datalog são compactas, se comparadas com as consultas SQL. Entretanto, quando as relações possuem um grande número de atributos, ou quando a ordem ou o número de atributos das relações pode mudar, a notação posicional pode ser desajeitada e propensa a erros. Não é difícil criar uma variante da sintaxe da Datalog usando atributos nomeados, em vez de atributos posicionais. Nesse sistema, a regra da Datalog definindo *v1* pode ser escrita como

v1(*número_conta A, saldo B*) :-
conta(*número_conta A, nome_agência* "Perryridge", *saldo B*),
B > 700

A tradução entre as duas formas pode ser feita sem um esforço significativo, dado o esquema de relação.

Sintaxe das regras Datalog

Agora que explicamos informalmente as regras e as consultas, podemos definir formalmente sua sintaxe; discutiremos seu significado na próxima seção. Usamos as mesmas convenções da álgebra relacional para indicar nomes de relação, nomes de atributo e constantes (como números ou strings entre aspas). Usamos letras maiúsculas e palavras com iniciais maiúsculas para indicar nomes de variáveis, e letras minúsculas e palavras com iniciais minúsculas para indicar nomes de relação e nomes de atributo. Exemplos de constantes são 4, que é um número, e "John", que é uma string; *X* e *Nome* são variáveis. Uma **literal positiva** tem a forma

$$p(t_1, t_2, ..., t_n)$$

onde p é o nome de uma relação com n atributos, e $t_1, t_2, ..., t_n$ são constantes ou variáveis. Uma **literal negativa** tem a forma

$$\textbf{not } p(t_1, t_2, ..., t_n)$$

onde a relação p possui n atributos. Aqui está um exemplo de uma literal:

$$conta(A, \text{"Perryridge"}, B)$$

Literais envolvendo operações aritméticas são tratadas de maneira especial. Por exemplo, a literal $B > 700$, embora não esteja na sintaxe descrita anteriormente, pode ser conceitualmente entendida como significando $>(B, 700)$, que *está* na sintaxe exigida, e onde $>$ é uma relação.

Todavia, o que essa notação significa para operações aritméticas como ">"? A relação > (conceitualmente) contém tuplas da forma (x, y) para cada par possível de valores x, y tal que $x > y$. Logo, (2, 1) e (5, –33) são tuplas em >. Claramente, a relação (conceitual) > é infinita. Outras operações aritméticas (tais como >, =, + e –) também são tratadas conceitualmente como relações. Por exemplo, A = B + C conceitualmente significa +(B, C, A), onde a relação + contém cada tupla (x, y, z) tal que $z = x + y$.

Um **fato** é escrito na forma

$$p(v_1, v_2, ..., v_n)$$

e indica que a tupla $(v_1, v_2, ..., v_n)$ está na relação p. Um conjunto de fatos para uma relação também pode ser escrito na notação tabular normal. Um conjunto de fatos para as relações em um esquema de banco de dados é equivalente a uma instância do esquema de banco de dados. As **regras** são construídas de literais e têm a forma

$$p(t_1, t_2, ..., t_n) :- L_1, L_2, ..., L_n$$

onde cada L_i é uma literal (positiva ou negativa). A literal $p(t_1, t_2, ..., t_n)$ é chamada o **cabeçalho** da regra e o restante das literais na regra constitui o **corpo** da regra.

Um **programa Datalog** consiste em um conjunto de regras; a ordem em que as regras são escritas não tem significância. Como dissemos anteriormente, pode haver várias regras definindo uma relação.

A Figura 5.7 mostra um programa Datalog que define os juros em cada conta na agência Perryridge. A primeira regra do programa define uma relação de view *juros*, cujos atributos são o número de conta e os juros recebidos na conta. Ela usa a relação *conta* e a relação de view *taxa_juros*. As duas últimas regras do programa são regras que vimos anteriormente.

Dizemos que uma relação view v_1 **depende diretamente** de uma relação de view v_2 se v_2 for usada na expressão definindo v_1. No programa exibido, a relação de view *juros* de-

```
juros(A, I) :- conta(A, "Perryridge", B)
               taxa_juros(A, R), I = B * R/100
taxa_juros(A, 5) :- conta(A, N, B), B < 10000
taxa_juros(A, 6) :- conta(A, N, B), B >= 10000
```

Figura 5.7 Programa Datalog que define juros sobre as contas em Perryridge.

$$func(X, Y) :- gerente(X, Y)$$
$$func(X, Y) :- gerente(X, Z), func(Z, Y)$$

Figura 5.8 Programa Datalog recursivo.

pende diretamente das relações *taxa_juros* e *conta*. A relação *taxa_juros*, por sua vez, depende diretamente de *conta*.

Dizemos que uma relação view v_1 **depende indiretamente de** uma relação de view v_2 se houver uma seqüência de relações intermediárias $i_1, i_2, ..., i_n$, para algum n, tal que v_1 dependa diretamente de i_1, i_1 dependa diretamente de i_2 e assim por diante até i_{n-1} dependendo de i_n.

No exemplo da Figura 5.7, já que temos uma cadeia de dependências indo de *juros* até *taxa_juros* e *conta*, a relação *juros* também depende indiretamente de *conta*.

Finalmente, dizemos que uma relação de view v_1 **depende da** relação de view v_2 se v_1 depender direta ou indiretamente de v_2.

Uma relação de view *v* é considerada **recursiva** se ela depender de si mesma. Uma relação de view que não é recursiva é chamada de **não recursiva**.

Considere o programa na Figura 5.8. Aqui, a relação de view *func* depende de si mesma (por causa da segunda regra), e, portanto, é recursiva. Por outro lado, o programa na Figura 5.7 é não recursivo.

Semântica da Datalog não recursiva

Consideramos a semântica formal dos programas Datalog. Por enquanto, consideramos apenas programas que são não recursivos. A semântica dos programas recursivos é um pouco mais complexa e é discutida na seção "Recursão na Datalog". Definimos a semântica de um programa começando com a semântica de uma única regra.

Semântica de uma regra

Uma **instanciação de base de uma regra** é o resultado da substituição de cada variável na regra por alguma constante. Se uma variável ocorre várias vezes em uma regra, todas as ocorrências da variável precisam ser substituídas pela mesma constante. As instanciações de base costumam ser chamadas simplesmente de **instanciações**.

Nossa regra de exemplo definindo *v1*, bem como uma instanciação da regra, são:

$$v1(A, B) :- conta(A, \text{"Perryridge"}, B), B > 700$$
$$v1(\text{"A-217"}, 750) :- conta(\text{"A-217"}, \text{"Perryridge"}, 750), 750 > 700$$

Aqui, a variável *A* foi substituída por "A-217" e a variável *B*, por 750.

Uma regra normalmente tem muitas instanciações possíveis. Essas instanciações correspondem a várias maneiras de atribuir valores a cada variável na regra.

Suponha que tenhamos uma regra *R*,

$$p(t_1, t_2, ..., t_n) :- L_1, L_2, ..., L_n$$

e um conjunto de fatos *I* para as relações usadas na regra (*I* também pode ser considerado uma instância de banco de dados). Considere qualquer instanciação *R'* da regra *R*:

$$p(v_1, v_2, ..., v_n) :- l_1, l_2, ..., l_n$$

onde cada literal, l_i é da forma $q_i(v_{i,1}, v_{i,2}, ... v_i\text{ni})$ ou da forma not $q_i(v_{i,1}, v_{i,2}, ... v_i\text{ni})$, e onde cada v_i e cada $v_{i,j}$ é uma constante.

Dizemos que o corpo da instanciação de regra *R'* é satisfeito em *I* se

1. Para cada literal positiva $q_i(v_{i,1}, ... v_i\text{ni})$ no corpo de *R'*, o conjunto de fatos *I* contém o fato $q(v_{i,1}, ... v_i\text{ni})$.
2. Para cada literal negativa $q_j(v_{j,1}, ... v_j\text{nj})$ no corpo de *R'*, o conjunto de fatos *I* não contém o fato $q_j(v_{j,1}, ... v_j\text{nj})$.

Definimos o conjunto de fatos que podem ser inferidos de um determinado conjunto de fatos *I* usando a regra *R* como

$$infer(R, I) = \{p(t_1, ..., t\text{ni}) \mid \text{há uma instanciação } R' \text{ de } R, \text{ onde } p(t_1, ..., t\text{ni}) \text{ é o cabeçalho de } R' \text{ e o corpo de } R' \text{ é satisfeito em } I\}.$$

Dado um conjunto de regras $\Re = \{R_1, R_2, ..., R_n\}$, definimos

$$infer(\Re, I) = infer(R_1, I) \cup infer(R_2, I) \cup ... \cup infer(R_n, I)$$

Suponha que temos um conjunto de fatos *I* contendo as tuplas para a relação *conta* na Figura 5.5. Uma instanciação possível de nossa regra de exemplo *R* é

$$v1(\text{"A-217"}, 750) :- conta(\text{"A-217"}, \text{"Perryridge"}, 750), 750 > 700$$

O fato *conta* ("A-217", "Perryridge", 750) está no conjunto de fatos *I*. Além disso, como 750 é maior que 700,

numero_conta	saldo
A-201	900
A-217	750

Figura 5.9 Resultado de *infer(R, I)*.

conceitualmente (750, 700) está na relação ">". Portanto, o corpo da instanciação de regra é satisfeito em *I*. Existem outras instanciações possíveis de *R* e, usando-as, descobrimos que *infer(R, I)* tem exatamente o conjunto de fatos para *vI* que aparece na Figura 5.9.

Semântica de um programa

Quando uma relação de view é definida em termos de outra relação de view, o conjunto de fatos na primeira view depende do conjunto de fatos da segunda view. Consideramos, nesta seção, que a definição é não recursiva; ou seja, nenhuma relação de view depende (direta ou indiretamente) de si mesma. Portanto, podemos dispor as relações de view em camadas da forma seguinte, e podemos usar essa disposição para definir a semântica do programa:

- Uma relação está na camada 1 se todas as relações usadas nos corpos das regras que a definem estiverem armazenadas no banco de dados.
- Uma relação está na camada 2 se todas as relações usadas nos corpos das regras que a definem estiverem armazenadas no banco de dados ou estiverem na camada 1.
- Em geral, uma relação *p* está na camada *i* + 1 se (1) ela não estiver nas camadas 1, 2,..., *i* e (2) todas as relações usadas nos corpos das regras que definem *p* estiverem armazenadas no banco de dados ou estiverem nas camadas 1, 2,..., *i*.

Considere o programa na Figura 5.7 com a regra adicional:

conta_perryridge(*X*, *Y*) :– *conta*(*X*, "Perryridge", *Y*)

A disposição das relações de view em camadas no programa aparece na Figura 5.10. A relação *conta* está no banco de dados. A relação *taxa_juros* está na camada 1, já que todas as relações usadas nas duas regras que a definem estão no banco de dados. Da mesma forma, a relação *conta_perryridge* está na camada 1. Finalmente, a relação *juros* está na camada 2, já que ela não está na camada 1 e todas as relações usadas na regra que a define estão no banco de dados ou nas camadas abaixo de 2.

Agora, podemos definir a semântica de um programa Datalog em termos da disposição das relações view em camadas. Façamos as camadas em um determinado programa serem 1, 2,..., *n*. Seja $\mathfrak{R}_i$ o conjunto de todas as regras que definem relações de view na camada *i*.

- Definimos I_0 para ser o conjunto dos fatos armazenados no banco de dados, e definimos I_1 como

$$I_1 = I_0 \cup infer(\mathfrak{R}_1, I_0)$$

- Continuamos de uma maneira semelhante, definindo I_2 em termos de I_1 e $\mathfrak{R}_2$ e assim por diante, usando a seguinte definição:

$$I_{i+1} = I_i \cup infer(\mathfrak{R}_{i+1}, I_i)$$

- Finalmente, o conjunto de fatos nas relações de view definidas pelo programa (também chamados **semântica do programa**) é dado pelo conjunto de fatos I_n correspondente à camada mais alta *n*.

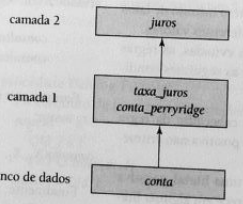


Figura 5.10 Disposição das relações view em camadas.

Para o programa na Figura 5.7, I_0 é o conjunto de fatos no banco de dados e I_1 é o conjunto de fatos no banco de dados juntamente com todos os fatos que podemos inferir de I_0 usando as regras para as relações *taxa_juros* e *conta_perryridge*. Finalmente, I_2 contém os fatos em I_1 juntamente com os fatos para a relação *juros* que podemos inferir dos fatos em I_1 pela regra definindo *juros*. A semântica do programa – ou seja, o conjunto dos fatos que estão em cada uma das relações de view – é definida como o conjunto de fatos I_2.

Lembre-se de que, na seção "Views definidas usando outras views" do Capítulo 3, vimos como definir o significado de views de álgebra relacional não recursivas por uma técnica conhecida como **expansão de view**. A expansão de view também pode ser usada com views de Datalog não recursivas; por outro lado, a técnica de disposição em camadas descrita aqui também pode ser usada com views de álgebra relacional.

Segurança

É possível escrever regras que geram um número infinito de respostas. Considere a regra

$$gt(X, Y) :- X > Y$$

Como a relação definindo > é infinita, essa regra geraria um número infinito de fatos para a relação *gt*, cujo cálculo, correspondentemente, levaria uma quantidade infinita de tempo e espaço.

O uso da negação também pode causar problemas semelhantes. Considere a regra

$$não\_no\_empréstimo(L, B, A) :- \text{not } empréstimo(L, B, A)$$

A idéia é que uma tupla (*número_empréstimo, nome_agência, quantia*) esteja na relação de view *não_no_empréstimo* se a tupla não estiver presente na relação *empréstimo*. Entretanto, se o conjunto de *números_empréstimo, nomes_agência* e *quantias* fosse infinito, a relação *não_no_empréstimo* também seria infinita.

Por fim, se tivermos uma variável no cabeçalho que não apareça no corpo, podemos ter um número infinito de fatos em que a variável é instanciada para diferentes valores.

Para que essas possibilidades sejam evitadas, as regras Datalog são necessárias para satisfazer as seguintes condições de **segurança**:

1. Cada variável que aparece no cabeçalho da regra também aparece em uma literal positiva não aritmética, no corpo da regra.
2. Cada variável aparecendo em uma literal negativa no corpo da regra também aparece em alguma literal positiva no corpo da regra.

Se todas as regras em um programa Datalog não recursivo satisfizerem essas condições de segurança, então, todas as relações de view definidas no programa podem ser mostradas como finitas, desde que todas as relações de banco de dados sejam finitas. As condições podem ser um pouco aliviadas para permitir que variáveis no cabeçalho apareçam apenas em uma literal aritmética no corpo em alguns casos. Por exemplo, na regra

$$p(A) :- q(B), A = B + 1$$

Podemos ver que se a relação q é finita, então, p também é finito, de acordo com as propriedades da adição, mesmo que a variável A apareça apenas em uma literal aritmética.

Operações relacionais na Datalog

As expressões Datalog não recursivas sem operações aritméticas são equivalentes em poder expressivo às expressões usando as operações básicas em álgebra relacional ($\cup$, $-$, $\times$, σ, Π e ρ). Não provaremos formalmente essa afirmação aqui. Em vez disso, mostraremos exemplos de como as várias operações de álgebra relacional podem ser expressas na Datalog. Em todos os casos, definimos uma relação de view chamada *consulta* para ilustrar as operações.

Já vimos como fazer seleção usando regras Datalog. Realizamos projeções simplesmente usando apenas os atributos exigidos no cabeçalho da regra. Para projetar o atributo *nome_conta* da conta, usamos

$$consulta(A) :- conta(A, N, B)$$

Podemos obter o produto cartesiano de duas relações r_1 e r_2 na Datalog desta maneira:

$$consulta(X_1, X_2, ..., X_n, Y_1, Y_2, ..., Y_m) :- r_1(X_1, X_2, ..., X_n), r_2(Y_1, Y_2, ..., Y_m)$$

onde r_1 é de aridade n e r_2 é de aridade m, e $X_1, X_2, ..., X_n, Y_1, Y_2, ..., Y_m$ são todos nomes de variável distintos.

Formamos a união de duas relações r_1 e r_2 (ambos de aridade n) da seguinte maneira:

$$consulta(X_1, X_2, ..., X_n) :- r_1(X_1, X_2, ..., X_n)$$
$$consulta(X_1, X_2, ..., X_n) :- r_2(X_1, X_2, ..., X_n)$$

Formamos a diferença de conjuntos das duas relações r_1 e r_2 assim:

$$consulta(X_1, X_2, ..., X_n) :- r_1(X_1, X_2, ..., X_n), \text{not } r_2(X_1, X_2, ..., X_n)$$

Finalmente, observamos que com a notação posicional usada na Datalog, o operador restante ρ não é necessário.

Uma relação pode ocorrer mais de uma vez no corpo da regra, mas, em vez de renomear para dar nomes distintos às ocorrências de relação, podemos usar nomes de variável diferentes nas diversas ocorrências.

É possível mostrar que podemos expressar qualquer consulta Datalog não recursiva sem aritmética usando operações de álgebra relacional. Deixamos essa demonstração como um exercício para você realizar. Portanto, você pode estabelecer a equivalência das operações básicas da álgebra relacional e a Datalog não recursiva sem operações aritméticas.

Certas extensões da Datalog aceitam as operações de atualização relacionais (inserção, exclusão e atualização). A sintaxe para essas operações varia de implementação para implementação. Alguns sistemas permitem o uso de + ou – nos cabeçalhos de regra para indicar inserção e exclusão relacional. Por exemplo, podemos mover todas as contas da agência Perryridge para a agência Johnstown executando

$$+conta(A, \text{"Johnstown"}, B) :- conta(A, \text{"Perryridge"}, B)$$
$$-conta(A, \text{"Perryridge"}, B) :- conta(A, \text{"Perryridge"}, B)$$

Algumas implementações da Datalog também aceitam a operação de agregação da álgebra relacional estendida. Novamente, não existe uma sintaxe padrão para essa operação.

Recursão na Datalog

Várias aplicações de banco de dados lidam com estruturas que são semelhantes às estruturas de dados de árvore. Por exemplo, considere os funcionários de uma organização. Alguns funcionários são gerentes. Cada gerente supervisiona um grupo de pessoas que estão subordinadas a ele. Contudo cada uma dessas pessoas, por sua vez, pode ser gerente e também pode ter outras pessoas subordinadas. Assim, os funcionários podem estar organizados em uma estrutura semelhante a uma árvore.

Suponha que temos um esquema de relação

$$Esquema\_gerente = (nome\_funcionario, nome\_gerente)$$

Façamos *gerente* ser uma relação nesse esquema.

Agora, suponha que desejamos descobrir que funcionários são supervisionados direta ou indiretamente por um determinado gerente – por exemplo, Jones. Portanto, se o gerente de Alon é Barinsky, o gerente de Barinsky é Estovar e o gerente de Estovar é Jones, então, Alon, Barinsky e Estovar são os funcionários controlados por Jones. As pessoas normalmente escrevem programas para manipular estruturas de dados por recursão. Usando a idéia de recursão, podemos definir o conjunto de funcionários controlados por Jones da maneira a seguir. As pessoas supervisionadas por Jones são (1) pessoas cujo gerente é Jones e (2) pessoas cujo gerente é supervisionado por Jones. Note que o caso (2) é recursivo.

Podemos codificar a definição recursiva anterior como uma view Datalog recursiva, chamada *func_jones*:

$$func\_jones(X) :- gerente(X, \text{"Jones"})$$
$$func\_jones(X) :- gerente(X, Y), func\_jones(Y)$$

A primeira regra corresponde ao caso (1); a segunda regra corresponde ao caso (2). A view *func_jones* depende de si mesma por causa da segunda regra; conseqüentemente, o programa Datalog anterior é recursivo. *Consideramos* que programas Datalog recursivos não contêm regras com literais negativos. A razão se tornará clara mais adiante. As notas bibliográficas fazem referência a documentos que descrevem onde a negação pode ser usada nos programas Datalog recursivos.

As relações de view de um programa recursivo que contém um conjunto de regras $\mathfrak{R}$ são definidas para conter exatamente o conjunto de fatos I calculado pelo procedimento iterativo Datalog-Fixpoint na Figura 5.11. A recursão no programa Datalog se transformou em uma repetição no procedimento. No final do procedimento, $infer(R_0, I) \cup D = I$, onde D é o conjunto de fatos no banco de dados e I é um **ponto fixo** do programa.

Considere o programa definindo *func_jones*, com a relação *gerente*, como na Figura 5.12. O conjunto de fatos calculado para a relação de view func_jones em cada repetição aparece na Figura 5.13. Em cada repetição, o programa cal-

```
procedure Datalog-Fixpoint
  I = conjunto de fatos no banco de dados
  repeat
    Old_I = I
    I = I ∪ infer(ℜ, I)
  until I = Old_I
```

Figura 5.11 Procedimento Datalog-Fixpoint.

nome_funcionário	nome_gerente
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Figura 5.12 Relação *gerente*.

Número da repetição	Tuplas em func_jones
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Anon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Anon)

Figura 5.13 Funcionários de Jones em repetições do procedimento Datalog-Fixpoint.

cula mais um nível de funcionários sob Jones e o acrescenta ao conjunto *func_jones*. O procedimento termina quando não há mais mudança no conjunto *func_jones*, o que o sistema detecta encontrando $I = Old\_I$. Esse ponto de término precisa ser atingido, já que o conjunto de gerentes e funcionários é finito. Na relação *gerente* dada, o procedimento Datalog-Fixpoint termina após a repetição 4, quando ele detecta que nenhum fato novo foi inferido.

Você deve verificar que, no final da repetição, a relação view *func_jones* contém exatamente os funcionários que trabalham para Jones. Para imprimir os nomes dos funcionários supervisionados por Jones definidos pela view, você pode usar a consulta

$$? \; func\_jones(N)$$

Para entender o procedimento Datalog-Fixpoint, devemos lembrar que uma regra infere novos fatos de um determinado conjunto de fatos. A repetição começa com um conjunto de fatos I definido para os fatos no banco de dados. Esses fatos são todos conhecidos como verdadeiros, mas pode haver outros que também o sejam.[1] A seguir, o conjunto de regras $\Re$ no programa Datalog dado é usado para inferir que fatos são verdadeiros, dado que os fatos em I são verdadeiros. Os fatos inferidos são acrescentados a I, e as regras são usadas novamente para fazer mais inferências. Esse processo é repetido até que nenhum fato novo possa ser inferido.

Para programas Datalog seguros, podemos mostrar que haverá algum ponto em que nenhum fato novo possa ser derivado; ou seja, para algum k, $I_{k+1} = I_k$. Nesse ponto, então, temos o conjunto final de fatos verdadeiros. Além disso, dado um programa Datalog e um banco de dados, o procedimento de ponto fixo infere todos os fatos que podem ser inferidos como verdadeiros.

Se um programa recursivo contém uma regra com uma literal negativa, o seguinte problema pode surgir. Lembre-se de que quando fazemos uma inferência usando uma instanciação de base de uma regra, para cada literal negativa not q no corpo da regra, verificamos que q não está presente no conjunto de fatos I. Esse teste considera que q não pode ser inferido mais tarde. Entretanto, na repetição de ponto fixo, o conjunto de fatos I cresce em cada repetição e, mesmo se q não estiver presente em I em uma repetição, ele pode aparecer em I mais tarde. Portanto, podemos ter feito uma inferência em uma repetição que pode não ser mais feita em uma repetição posterior e a inferência foi incorreta. Exigimos que um programa recursivo não deve conter literais negativas, para evitar esses problemas.

Em vez de criar uma view para os funcionários supervisionados por um gerente específico, Jones, podemos criar uma relação de view mais geral, *func*, que contenha cada tu-

[1] A palavra "fato" é usada em um sentido técnico para notar a participação de uma tupla em uma relação. Portanto, no sentido da Datalog de "fato", um fato pode ser verdadeiro (a tupla está realmente na relação) ou falso (a tupla não está na relação).

pla (X, Y) tal que X seja direta ou indiretamente gerenciado por Y, usando o seguinte programa (também mostrado na Figura 5.8):

$$func(X, Y) :- gerente(X, Y)$$
$$func(X, Y) :- gerente(X, Z), func(Z, Y)$$

Para encontrar os subordinados diretos e indiretos de Jones, simplesmente usamos a consulta

$$? func(X, \text{"Jones"})$$

que fornece o mesmo conjunto de valores para X que a view *func_jones*. A maioria das implementações da Datalog possui otimizadores de consulta e mecanismos de avaliação sofisticados que podem executar a consulta anterior praticamente na mesma velocidade que poderiam avaliar a view *func_jones*.

A view *func* definida anteriormente é o **fechamento transitivo** da relação *gerente*. Se a relação *gerente* fosse substituída por qualquer outra relação binária R, o programa anterior definiria o fechamento transitivo de R.

O poder da recursão

A Datalog com recursão possui mais poder expressivo do que a Datalog sem recursão. Em outras palavras, existem consultas no banco de dados que podemos responder usando recursão, mas não podemos responder sem usá-la. Por exemplo, não podemos expressar fechamento transitivo na Datalog sem usar recursão (ou na SQL ou QBE sem recursão). Considere o fechamento transitivo da relação *gerente*. Intuitivamente, um número fixo de junções pode encontrar apenas os funcionários que estão a algum (outro) número fixo de níveis abaixo de qualquer gerente (não tentaremos provar esse resultado aqui). Como qualquer consulta não recursiva tem um número fixo de junções, existe um limite em relação a quantos níveis de funcionários a consulta pode encontrar. Se o número de níveis de funcionários na relação *gerente* for maior do que o limite da consulta, a consulta omitirá alguns níveis de funcionários. Portanto, um programa Datalog não recursivo não pode expressar fechamento transitivo.

Uma alternativa à recursão é usar um mecanismo externo, como a SQL embutida, para repetir em uma consulta não recursiva. A repetição efetivamente implementa o loop de ponto fixo da Figura 5.11. Na verdade, é desse modo que as consultas são implementadas em sistemas de banco de dados que não aceitam recursão. Entretanto, escrever essas consultas por repetição é mais complicado do que usar recursão, e a avaliação por recursão pode ser otimizada para ser executada mais rápido do que a avaliação por repetição.

O poder expressivo fornecido pela recursão precisa ser usado com cuidado. É relativamente fácil escrever programas recursivos que gerarão um número infinito de fatos, como este programa ilustra:

$$número(0)$$
$$número(A) :- número(B), A = B + 1$$

O programa gera *número*(n) para todos os inteiros positivos n, que é claramente infinito, e não terminará. A segunda regra do programa não satisfaz a condição de segurança na seção "Segurança". Os programas que satisfazem a condição de segurança terminarão, mesmo se eles forem recursivos, desde que todas as relações de banco de dados forem finitas. Para esses programas, as tuplas nas relações de view podem conter apenas constantes do banco de dados e, portanto, as relações de view precisam ser finitas. O contrário não é verdade; ou seja, existem programas que não satisfazem as condições de segurança, mas que terminam.

O procedimento Datalog-Fixpoint usa repetidamente a função $infer(R_v, I)$ para calcular quais fatos são verdadeiros, dado um programa Datalog recursivo. Embora consideremos apenas o caso de programas Datalog sem literais negativas, o procedimento também pode ser usado em views definidas em outras linguagens, como SQL ou álgebra relacional, contanto que as views satisfaçam as condições descritas a seguir. Seja qual for a linguagem usada para definir uma view V, a view pode ser imaginada como sendo definida por uma expressão E_v que, dado um conjunto de fatos I, retorna um conjunto de fatos $E_v(I)$ para a relação de view V. Dado um conjunto de definições de view R_v (em qualquer linguagem), podemos definir uma função $infer(R_v, I)$ que retorna $I \cup \bigcup_{v \in R_v} E_v(I)$. A função anterior tem a mesma forma da função *infer* para Datalog.

Dizemos que uma view V é **monotônica** se, dados dois conjuntos quaisquer de fatos, I_1 e I_2 tais que $I_1 \subseteq I_2$, então, $E_v(I_1) \subseteq E_v(I_2)$, onde E_v é a expressão usada para definir V. Da mesma forma, dizemos que a função *infer* é monotônica se

$$I_1 \subseteq I_2 \Rightarrow infer(R_v, I_1) \subseteq infer(R_v, I_2)$$

Portanto, se *infer* for monotônica, dado um conjunto de fatos I_0 que seja um subconjunto dos fatos verdadeiros, podemos ter certeza de que todos os fatos em $infer(R_v, I_0)$ também são verdadeiros. Usando o mesmo raciocínio da seção anterior, podemos, então, mostrar que o procedimento Datalog-Fixpoint é confiável (ou seja, calcula apenas fatos verdadeiros), desde que a função *infer* seja monotônica.

As expressões de álgebra relacional que usam apenas os operadores Π, σ, $\times$, $\bowtie$, $\cup$, $\cap$ ou ρ são monotônicas. As views recursivas podem ser definidas usando essas expressões.

Entretanto, as expressões relacionais que usam o operador – não são monotônicas. Por exemplo, sejam $gerente_1$ e $gerente_2$ relações com o mesmo esquema da relação *gerente*. Dados

I_1 = {$gerente_1$("Alon", "Barinsky"), $gerente_1$("Barinsky", "Estovar"), $gerente_2$("Alon", "Barinsky")}

e

I_2 = {$gerente_1$("Alon", "Barinsky"), $gerente_1$("Barinsky", "Estovar"), $gerente_2$("Alon", "Barinsky"), $gerente_2$("Barinsky", "Estovar")}

considere a expressão $gerente_1 - gerente_2$. Agora, o resultado da expressão anterior em I_1 é ("Barinsky", "Estovar"), enquanto o resultado da expressão em I_2 é a relação vazia. Todavia, como $I_1 \subseteq I_2$, então, a expressão não é monotônica. As expressões usando a operação de agrupamento da álgebra relacional estendida também não são monotônicas.

A técnica do ponto fixo não funciona em views recursivas definidas com expressões não monotônicas. Entretanto, há instâncias em que essas views são úteis, especialmente para definir agregadas em relações "parte-subparte". Essas relações definem que subpartes compõem cada parte. As próprias subpartes podem ter mais subpartes e assim por diante; portanto, as relações, como a relação *gerente*, possuem uma estrutura recursiva natural. Um exemplo de uma consulta agregada nessa estrutura seria calcular o número total de subpartes de cada parte. Escrever essa consulta em Datalog ou SQL (sem extensões procedurais) exigiria o uso de uma view recursiva em uma expressão não monotônica. As notas bibliográficas fornecem referências para pesquisa sobre a definição dessas views.

É possível definir alguns tipos de consultas recursivas sem usar views. Por exemplo, foram propostas operações relacionais estendidas para definir fechamento transitivo, e também foram propostas extensões à sintaxe da SQL para especificar fechamento transitivo (generalizado). No entanto, as definições de view recursivas fornecem mais poder expressivo do que outras formas de consultas recursivas.

Resumo

- O **cálculo relacional de tupla** e o **cálculo relacional de domínio** são linguagens não procedurais que representam a capacidade básica necessária em uma linguagem de consulta relacional. A álgebra relacional básica é uma linguagem procedural que possui poder expressivo equivalente a ambas as formas do cálculo relacional quando estão restritas a expressões seguras.
- Os cálculos relacionais são linguagens concisas e formais, que são impróprias para usuários casuais de um sistema de banco de dados. Os sistemas de banco de dados comerciais, portanto, usam linguagens com mais "açúcar sintático". Consideramos duas linguagens de consulta: QBE e Datalog.
- A QBE é baseada em um modelo visual: as consultas se parecem muito com tabelas.
- A QBE e suas variantes se tornaram atraentes para usuários de banco de dados não especialistas devido à simplicidade intuitiva do modelo visual. O amplamente usado sistema de banco de dados Microsoft Access aceita uma versão gráfica da QBE, chamada GQBE.
- A Datalog é derivada da Prolog, mas, diferente da Prolog, ela possui uma semântica declarativa, tornando as consultas simples mais fáceis de escrever e a avaliação de consulta mais fácil de otimizar.
- É particularmente fácil definir views na Datalog. Além disso, as views recursivas que a Datalog aceita possibilitam escrever consultas, como consultas de fechamento transitivo, que não podem ser escritas sem recursão ou repetição. Entretanto, não existe na Datalog um padrão aceito para recursos importantes, como agrupamento e agregação. A Datalog permanece sendo principalmente uma linguagem de pesquisa.

Termos de revisão

- Cálculo relacional de tupla
- Cálculo relacional de domínio
- Segurança de expressões
- Poder expressivo das linguagens
- Query-by-Example (QBE)
- Sintaxe bidimensional
- Tabelas de estrutura
- Linhas de exemplo
- Caixa de condição
- Relação resultado
- Microsoft Access
- Graphical Query-By-Example (GQBE)
- Grade de projeto
- Datalog
- Regras
- Usa
- Define
- Literal positiva
- Literal negativa
- Fato
- Regra
  - Cabeçalho
  - Corpo
- Programa Datalog

- Depender de
  - ☐ Diretamente
  - ☐ Indiretamente
- Visão recursiva
- Visão não recursiva
- Instanciação
  - ☐ Instanciação de base
  - ☐ Satisfeita
- Inferência
- Semântica
  - ☐ De uma regra
  - ☐ De um programa
- Segurança
- Ponto fixo
- Fechamento transitivo
- Definição de view monotônica

Exercícios práticos

5.1 Sejam os seguintes esquemas de relação:

$$R = (A, B, C)$$
$$S = (D, E, F)$$

Sejam $r(R)$ e $s(S)$ as relações dadas. Forneça uma expressão no cálculo relacional de tupla que seja equivalente a cada um dos seguintes:

a. $\Pi_A(r)$
b. $\sigma_{B = 17}(r)$
c. $r \times s$
d. $\Pi_{A,F}(\sigma_{C = D}(r \times s))$

5.2 Seja $R = (A, B, C)$ e sejam r_1 e r_2 relações no esquema R. Forneça uma expressão no cálculo relacional de domínio que seja equivalente a cada um dos seguintes:

a. $\Pi_A(r_1)$
b. $\sigma_{B = 17}(r_1)$
c. $r_1 \cup r_2$
d. $r_1 \cap r_2$
e. $r_1 - r_2$
f. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

5.3 Seja $R = (A, B)$ e $S = (A, C)$, e sejam $r(R)$ e $s(S)$ relações. Escreva expressões em QBE e Datalog para cada uma das seguintes consultas:

a. $\{< a > \mid \exists b\,(< a, b > \in r \wedge b = 7)\}$
b. $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$
c. $\{< a > \mid \exists c\,(< a, c > \in s \wedge \exists b_1, b_2\,(< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

5.4 Considere o banco de dados relacional da Figura 5.14, no qual as chaves primárias estão sublinhadas. Forneça uma expressão em Datalog para cada uma das seguintes consultas:

a. Encontre todos os funcionários que trabalham (direta ou indiretamente) para o gerente "Jones".
b. Encontre todas as cidades de residência de todos os funcionários que trabalham (direta ou indiretamente) para o gerente "Jones".
c. Encontre todos os pares de funcionários que possuem um gerente (direto ou indireto) em comum.
d. Encontre todos os pares de funcionários que possuem um gerente (direto ou indireto) em comum, e que se encontram no mesmo número de níveis de supervisão abaixo do gerente em comum.

5.5 Descreva como uma regra Datalog arbitrária pode ser expressa como uma view de álgebra relacional estendida.

Exercícios

5.6 Considere o banco de dados de funcionários da Figura 5.14. Forneça expressões em cálculo relacional de tupla e em cálculo relacional de domínio para cada uma das seguintes consultas:

a. Encontre os nomes de todos os funcionários que trabalham para o First Bank Corporation.
b. Encontre os nomes e cidades de residência de todos os funcionários que trabalham para o First Bank Corporation.
c. Encontre os nomes, ruas e cidades de todos os funcionários que trabalham para o First Bank Corporation e ganham mais de $10.000 por ano.
d. Encontre os nomes de todos os funcionários que moram na mesma cidade onde está localizada a empresa para a qual trabalham.

funcionário (<u>nome_pessoa</u>, *rua*, *cidade*)
trabalha (<u>nome_pessoa</u>, *nome_empresa*, *salário*)
empresa (<u>nome_empresa</u>, *cidade*)
gerência (<u>nome_pessoa</u>, *nome_gerente*)

Figura 5.14 Banco de dados de funcionários.

e. Encontre os nomes de todos os funcionários que moram na mesma cidade e na mesma rua de seus gerentes.

f. Encontre os nomes de todos os funcionários no banco de dados que não trabalham para o First Bank Corporation.

g. Encontre os nomes de todos os funcionários que ganham mais do que todos os funcionários do Small Bank Corporation.

h. Considerando que as empresas podem estar localizadas em várias cidades, encontre todas as empresas em cada cidade onde o Small Bank Corporation está localizado.

5.7 Sejam $R = (A, B)$ e $S = (A, C)$, e sejam $r(R)$ e $s(S)$ relações. Escreva expressões de álgebra relacional equivalentes às seguintes expressões de cálculo relacional de domínio:

a. $\{< a > \mid \exists\, b\, (< a, b > \in r \wedge b = 17)\}$

b. $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$

c. $\{< a > \mid \exists\, b\, (< a, b > \in r) \vee \forall\, c\, (\exists\, d\, (< d, c > \in s) \Rightarrow < a, c > \in s)\}$

d. $\{< a > \mid \exists\, c\, (< a, c > \in s \wedge \exists\, b_1, b_2\, (< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

5.8 Repita o Exercício 5.7, escrevendo consultas SQL em vez de expressões de álgebra relacional.

5.9 Sejam $R = (A, B)$ e $S = (A, C)$, e sejam $r(R)$ e $s(S)$ relações. Usando a constante especial *nulo*, escreva expressões de cálculo relacional de tupla equivalentes a cada um dos seguintes:

a. $r \ltimes s$

b. $r \rtimes s$

c. $r \rtimes\!\ltimes s$

5.10 Considere o banco de dados de seguros da Figura 5.15, no qual as chaves primárias estão sublinhadas. Construa as seguintes consultas GQBE para esse banco de dados relacional.

a. Encontre o número total de pessoas que possuíram carros que foram envolvidos em acidentes em 1989.

b. Encontre o número de acidentes em que os carros pertencentes a "John Smith" estavam envolvidos.

5.11 Forneça uma expressão de cálculo relacional de tupla para encontrar o valor máximo na relação $r(A)$.

5.12 Repita o Exercício 5.6 usando a QBE e a Datalog.

5.13 Seja $R = (A, B, C)$ e sejam r_1 e r_2 relações no esquema R. Forneça expressões em QBE e Datalog equivalentes a cada uma das seguintes consultas:

a. $r_1 \cup r_2$

b. $r_1 \cap r_2$

c. $r_1 - r_2$

d. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

5.14 Escreva uma view de álgebra relacional estendida equivalente à regra Datalog

$$p(A, C, D) :- q1(A, B), q2(B, C), q3(A, B), D = B + 1$$

Notas bibliográficas

A definição original de cálculo relacional de tupla está em Codd [1972]. Uma prova formal da equivalência entre o cálculo relacional de tupla e a álgebra relacional está em Codd [1972]. Várias extensões ao cálculo relacional foram propostas. Klug [1982] e Escobar-Molano *et al.* [1993] descrevem extensões às funções agregadas escalares.

O sistema de banco de dados QBE foi desenvolvido no T. J. Watson Research Center da IBM no início da década de 1970. A linguagem de manipulação de dados QBF foi posteriormente usada no Query Management Facility (QMF) da IBM. A versão original da Query-by-Example é descrita em Zloof [1977]. Outras implementações da QBE incluem o Microsoft Access e o Borland Paradox (que não é mais aceito).

Ullman [1988] e Ullman [1989] fornecem discussões de texto amplas sobre as linguagens de consulta lógicas e técnicas de implementação. Ramakrishnan e Ullman [1995] fornecem uma pesquisa mais recente sobre os bancos de dados dedutivos.

Uma semântica simples pode ser atribuída aos programas Datalog que possuem recursão e negação se a negação for "estratificada" – ou seja, se não houver recursão por negação. Chandra e Harel [1982] e Apt e Pugin [1987] discutem a negação estratificada. Uma importante extensão chamada *semântica de estratificação modular*, que manipula

pessoa (id_motorista, nome, endereço)
carro (licença, modelo, ano)
acidente (número_ocorrência, data, local)
pertence (id_motorista, licença)
participou (id_motorista, carro, número_ocorrência, valor_dano)

Figura 5.15 Banco de dados de seguros.

uma classe de programas recursivos com literais negativas, é discutida em Ross [1990]; uma técnica de avaliação para esses programas é descrita por Ramakrishnan e outros [1992].

Ferramentas

O Microsoft Access QBE é atualmente a implementação mais amplamente disponível da QBE. As edições QMF e Everywhere do IBM DB2 também possuem suporte para a QBE.

O sistema Coral da University of Wisconsin-Madison (`www.cs.wisc.edu/coral`) é uma implementação da Datalog. O sistema XSB da State University of New York (SUNY) Stony Brook (`xsb.sourceforge.net`) é uma implementação da Prolog largamente utilizada, que aceita consultas de banco de dados; lembre-se de que Datalog é um subconjunto não procedural da Prolog.

PARTE 2

Projeto de banco de dados

Os sistemas de banco de dados são projetados para gerenciar grandes blocos de informação. Esses grandes blocos de informação não existem isolados. Eles são parte da operação de alguma empresa cujo produto final pode ser informações do banco de dados ou pode ser algum dispositivo ou serviço para o qual o banco de dados desempenha um papel de apoio.

Os dois primeiros capítulos desta parte focalizam o projeto dos esquemas de banco de dados. O modelo entidade-relacionamento (E-R), descrito no Capítulo 6, é um modelo de dados de alto nível. Em vez de representar todos os dados em tabelas, esse modelo distingue entre objetos básicos, chamados *entidades*, e *relacionamentos* entre esses objetos. Ele normalmente é usado como uma primeira etapa no projeto de esquema de banco de dados.

O projeto de banco de dados relacional – o projeto do esquema relacional – foi discutido informalmente em capítulos anteriores. Existem, no entanto, princípios que podem ser usados para distinguir entre bons e maus projetos de banco de dados. Esses são formalizados por meio de várias "formas normais" que oferecem diferentes compensações entre a possibilidade de inconsistências e a eficiência de certas consultas. O Capítulo 7 descreve o projeto formal dos esquemas relacionais.

O projeto de um ambiente de aplicação de banco de dados completo, que atende às necessidades da empresa sendo modelada, requer atenção para um conjunto mais amplo de aspectos, muitos dos quais são abordados no Capítulo 8. Esse capítulo descreve interfaces baseadas na Web a bancos de dados e estende nossa discussão anterior sobre integridade e segurança de dados.

CAPÍTULO 6

Projeto de banco de dados e o modelo E-R

Até este ponto no livro, consideramos um determinado esquema de banco de dados e estudamos como as consultas e atualizações são expressas. Agora, consideraremos, em primeiro lugar, como projetar um esquema de banco de dados. Neste capítulo, focalizaremos o modelo de dados entidade-relacionamento (E-R), que fornece um meio de identificar entidades a serem representadas no banco de dados e como essas entidades são relacionadas. Finalmente, o projeto de banco de dados será expresso em termos de um projeto de banco de dados relacional e um conjunto associado de restrições. Mostramos, neste capítulo, como um projeto E-R pode ser transformado em um conjunto de esquemas de relação e como algumas das restrições podem ser capturadas nesse projeto. Depois, no Capítulo 7, consideraremos em detalhes se um conjunto de esquemas de relação representa um bom ou mau projeto de banco de dados e estudaremos o processo de criar bons projetos usando um conjunto mais amplo de restrições. Esses dois capítulos abordam os conceitos fundamentais do projeto de banco de dados.

Visão geral do processo de projeto

A criação de uma aplicação de banco de dados é uma tarefa complexa, que envolve várias tarefas, como o projeto do esquema de banco de dados, o projeto dos programas que acessam e atualizam os dados e o projeto de um esquema de segurança para controlar o acesso aos dados. As necessidades dos usuários desempenham um papel central no processo de projeto. Neste capítulo, focalizamos o projeto do esquema de banco de dados, embora descrevamos brevemente algumas outras tarefas de projeto mais adiante no capítulo.

O projeto de um ambiente de aplicação de banco de dados completo que atenda às necessidades da empresa sendo modelada requer atenção a um amplo conjunto de aspectos. Esses aspectos adicionais do uso esperado do banco de dados influenciam uma variedade de escolhas de projeto nos níveis físico, lógico e visual.

Fases do projeto

Para pequenas aplicações, pode ser viável para um projetista de banco de dados que conheça as necessidades da aplicação decidir diretamente sobre as relações a serem criadas, seus atributos e as restrições nas relações. Entretanto, esse processo de projeto direto é difícil para aplicações do mundo real, já que, em geral, elas são altamente complexas. Normalmente, nenhuma pessoa entende todas as necessidades de dados de uma aplicação. O projetista de banco de dados precisa interagir com os usuários da aplicação para entender as necessidades da aplicação, representá-las de uma maneira de alto nível que possa ser entendida pelos usuários e, depois, traduzir as necessidades para níveis mais baixos do projeto. O modelo de dados de alto nível serve ao projetista de banco de dados fornecendo uma estrutura conceitual na qual ele pode especificar, de um modo sistemático, as necessidades de dados dos usuários do banco de dados e uma estrutura de banco de dados que satisfaça essas necessidades.

- A fase inicial do projeto de banco de dados é caracterizar completamente as necessidades de dados dos prováveis usuários do banco de dados. O projetista de banco de dados precisa interagir extensivamente com especialistas de domínio e usuários para realizar essa tarefa. A saí-

da dessa fase é uma especificação das necessidades do usuário. Embora existam técnicas para representar graficamente as necessidades do usuário, neste capítulo iremos nos preocupar com as descrições textuais das necessidades do usuário, que ilustraremos mais adiante, na seção "Necessidades de projeto para o banco de dados de banco".

- Em seguida, o projetista escolhe um modelo de dados e, aplicando os conceitos do modelo de dados escolhido, traduz essas necessidades para um esquema conceitual do banco de dados. O esquema desenvolvido nessa fase de **projeto conceitual** fornece uma descrição detalhada da empresa. O modelo entidade-relacionamento, que estudaremos no restante deste capítulo, normalmente é usado para representar o projeto conceitual. Em termos de modelo entidade-relacionamento, o esquema conceitual especifica as entidades que são representadas no banco de dados, os atributos das entidades, os relacionamentos entre as entidades e as restrições sobre as entidades. Em geral, a fase do projeto conceitual resulta na criação de um diagrama de entidade-relacionamento que fornece uma representação gráfica do esquema.

  O projetista verifica o esquema para confirmar se todas as necessidades de dados são realmente satisfeitas e não estão em conflito umas com as outras. Ele também pode examinar o projeto para remover quaisquer recursos redundantes. Sua preocupação nesse momento é descrever os dados e seus relacionamentos em vez de especificar detalhes de armazenamento físico.
- Um esquema conceitual completamente desenvolvido também indica as necessidades funcionais da empresa. Em uma **especificação das necessidades funcionais**, os usuários descrevem os tipos de operações (ou transações) que serão realizadas nos dados. Operações de exemplo incluem modificar ou atualizar dados, pesquisar e recuperar dados específicos e excluir dados. Nessa fase do projeto conceitual, o projetista pode revisar o esquema para garantir que ele atenda às necessidades funcionais.
- O processo de transição de um modelo de dados abstrato o para a implementação do banco de dados ocorre nas duas fases finais do projeto.
  - Na **fase de projeto lógico**, o projetista mapeia o esquema conceitual de alto nível para o modelo de dados de implementação do sistema de banco de dados que será usado. O modelo de dados de implementação normalmente é o modelo de dados relacional, e essa etapa geralmente consiste em mapear o esquema conceitual definido usando o modelo entidade-relacionamento em um esquema de relação.
  - Finalmente, o projetista usa o esquema de banco de dados específico do sistema resultante na **fase de projeto físico**, em que os recursos físicos do banco de dados são especificados. Esses recursos incluem a forma de organização de arquivo e as estruturas de armazenamento internas; eles são discutidos no Capítulo 11.

O esquema físico de um banco de dados pode ser modificado de modo relativamente fácil após uma aplicação ter sido construída. Entretanto, as mudanças no esquema lógico normalmente são mais difíceis de realizar, já que podem afetar diversas consultas e atualizações disseminadas pelo código da aplicação. Portanto, é importante realizar a fase de projeto de banco de dados com cuidado, antes de construir o restante da aplicação de banco de dados.

Alternativas de projeto

Uma parte importante do processo de projeto de banco de dados é decidir como representar no projeto os vários tipos de "coisas", como pessoas, lugares, produtos e assim por diante. Usamos o termos *entidade* para representar qualquer item distintamente identificável. Essas várias entidades possuem certas peculiaridades, bem como diferenças. Desejamos explorar as peculiaridades para ter um projeto sucinto e facilmente entendido, mas precisamos manter a flexibilidade em representar distinções entre entidades que existem em tempo de projeto ou que podem se materializar no futuro. As várias entidades estão inter-relacionadas de diversas maneiras, todas precisando ser capturadas no projeto de banco de dados.

1. **Redundância**: um mau projeto pode repetir informações. No exemplo de banco que usamos até agora, temos uma relação com informações de cliente e uma relação separada com informações de conta. Suponha que, em vez disso, repetíssemos todas as informações de cliente (nome, endereço etc.) uma vez para cada conta ou empréstimo que o cliente possui. Claramente, isso seria redundante. Idealmente, as informações devem aparecer exatamente em um lugar.
2. **Falta de integralidade**: um mau projeto pode tornar certos aspectos da empresa difíceis ou impossíveis de ser modelados. Por exemplo, suponha que usássemos um projeto de banco de dados para nosso cenário de banco que armazene informações de nome e endereço de cliente com cada conta e empréstimo, mas não tenha uma relação cliente separada. Então, seria impossível inserir o nome e o endereço de um novo cliente a menos que esse cliente já tivesse uma conta aberta ou feito um empréstimo no banco. Poderíamos tentar nos conformar com o projeto problemático armazenando valores nulos para conta ou

informações de empréstimo, como número de conta ou quantia. Essa solução não só é desagradável, mas pode ser evitada por restrições de chave primária.

Evitar maus projetos não é o bastante. Pode haver um grande número de bons projetos que precisamos escolher. Como um exemplo simples, considere um cliente que compra um produto. A venda desse produto é um relacionamento entre o cliente e o produto? Alternativamente, é a própria venda uma entidade que está relacionada com o cliente e com o produto? Essa escolha, embora simples, pode fazer uma importante diferença em que aspectos da empresa podem ser bem modelados. Considerando a necessidade de fazer escolhas como essa para o grande número de entidades e relacionamentos em uma empresa real, não é difícil ver que o projeto de banco de dados pode ser um problema complexo. Sem dúvida, veremos que ele requer uma combinação de ciência e "bom-gosto".

O modelo entidade-relacionamento

O modelo de dados **entidade-relacionamento** (E-R) foi desenvolvido para facilitar o projeto de banco de dados, permitindo especificação de um *esquema de empresa* que representa a estrutura lógica geral de um banco de dados. O modelo E-R é muito útil no mapeamento dos significados e interações de empresas reais para um esquema conceitual. Devido a essa utilidade, muitas ferramentas de projeto de banco de dados utilizam conceitos do modelo E-R. O modelo de dados E-R emprega três noções básicas: conjuntos de entidades, conjuntos de relacionamento e atributos.

Conjuntos de entidades

Uma **entidade** é uma "coisa" ou "objeto" no mundo real, que é distinguível de todos os outros objetos. Por exemplo, cada pessoa em uma empresa é uma entidade. Uma entidade possui um conjunto de propriedades, e os valores para algum conjunto de propriedades podem identificar de maneira única uma entidade. Assim, uma pessoa pode ter uma propriedade *id_pessoa* cujo valor identifica unicamente essa pessoa. Portanto, o valor 677-89-9011 para o *id_pessoa* identifica unicamente uma determinada pessoa na empresa. Da mesma forma, os empréstimos podem ser considerados como entidades, e o número de empréstimo L-15 na agência Perryridge identifica unicamente uma entidade empréstimo. Uma entidade pode ser concreta, como uma pessoa ou um livro, ou pode ser abstrata, como um empréstimo, um feriado ou um conceito.

Um **conjunto de entidades** é um conjunto de entidades do mesmo tipo que compartilham as mesmas propriedades ou atributos. O conjunto de todas as pessoas que são clientes de um determinado banco, por exemplo, pode ser definido como o conjunto de entidades *cliente*. De igual modo, o conjunto de entidades *empréstimo* pode representar o conjunto de todos os empréstimos concedidos por um determinado banco. As entidades individuais que constituem um conjunto são chamadas a *extensão* do conjunto de entidades. Assim, todos os clientes de banco individuais são a extensão do conjunto de entidades *cliente*.

Os conjuntos de entidades não precisam ser desconexos. Por exemplo, é possível definir o conjunto de entidades de todos os funcionários de um banco (*funcionário*) e o conjunto de entidades de todos os clientes do banco (*cliente*). Uma entidade *pessoa* pode ser uma entidade *funcionário*, uma entidade *cliente*, as duas coisas ou nenhuma delas.

Uma entidade é representada por um conjunto de **atributos**. Os atributos são propriedades descritivas possuídas por membro de um conjunto de entidades. A designação de um atributo para um conjunto de entidades expressa que o banco de dados armazena informações semelhantes concernentes a cada entidade no conjunto de entidades; entretanto, cada entidade pode ter seu próprio valor para cada atributo. Os possíveis atributos do conjunto de entidades *cliente* são *id_cliente*, *nome_cliente*, *rua_cliente* e *cidade_cliente*. Na vida real, haveria outros atributos, como número de rua, número de apartamento, estado, código postal e país, mas foram omitidos para simplificar nossos exemplos. Os possíveis atributos do conjunto de entidades *empréstimo* são *número_empréstimo* e *quantia*.

Cada entidade possui um **valor** para cada um de seus atributos. Por exemplo, uma determinada entidade cliente pode ter o valor 321-12-3123 para *id_cliente*, o valor Jones para *nome_cliente*, o valor Main para *rua_cliente* e o valor Harrison para *cidade_cliente*.

O atributo *id_cliente* é usado para identificar clientes de maneira única, já que pode haver mais de um cliente com o mesmo nome, rua e cidade. Nos Estados Unidos, muitas empresas acham conveniente usar o número de *seguro_social* de uma pessoa[1] como um atributo cujo valor identifica unicamente a pessoa. Em geral, a empresa precisa criar e atribuir um identificador único para cada cliente.

Portanto, um banco de dados inclui uma coleção de conjuntos de entidades, cada um contendo qualquer número de entidades do mesmo tipo. A Figura 6.1 mostra parte de um banco de dados de um banco que consiste em dois conjuntos de entidades: *cliente* e *empréstimo*.

1. Nos Estados Unidos, o governo atribui a cada pessoa do país um número único, chamado número de seguro social, para identificar essa pessoa unicamente. Cada pessoa deve ter apenas um número de seguro social e nenhuma pessoa deve ter o mesmo número de seguro social de outra pessoa.

321-12-3123	Jones	Main	Harrison
019-28-3746	Smith	North	Rye
677-89-9011	Hayes	Main	Harrison
555-55-5555	Jackson	Dupont	Woodside
244-66-8800	Curry	North	Rye
963-96-3963	Williams	Nassau	Princeton
335-57-7991	Adams	Spring	Pittsfield

cliente

L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-19	500
L-11	900
L-16	1300

empréstimo

Figura 6.1 Conjuntos de entidades *cliente* e *empréstimo*.

Um banco de dados para uma instituição bancária pode incluir inúmeros outros conjuntos de entidades. Por exemplo, além de controlar os clientes e empréstimos, o banco também fornece contas, que são representadas pelo conjunto de entidades *conta* com atributos *número_conta* e *saldo*. Além disso, se o banco tiver muitas agências diferentes, poderemos manter informações sobre todas as suas agências. Cada conjunto de entidades *agência* pode ser descrito pelos atributos *nome_agência*, *cidade_agência* e *ativo*.

Conjuntos de relacionamento

Um **relacionamento** é uma associação entre várias entidades. Por exemplo, podemos definir um relacionamento que associa o cliente Hayes com o empréstimo L-15. Esse relacionamento especifica que Hayes é um cliente com o número de empréstimo L-15.

Um **conjunto de relacionamento** é um conjunto de relacionamentos do mesmo tipo. Formalmente, ele é uma relação matemática em conjuntos de entidades $n \geq 2$ (possivelmente indistintos). Se $E_1, E_2, ..., E_n$ são conjuntos de entidades, então, um conjunto de relacionamento R é um subconjunto de

$$\{(e_1, e_2, ..., e_n) \mid e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n\}$$

onde $(e_1, e_2, ..., e_n)$ é um relacionamento.

Considere os dois conjuntos de entidades *cliente* e *empréstimo* da Figura 6.1. Definimos o conjunto de relacionamento *tomador* para indicar a associação entre os clientes e os empréstimos bancários que os clientes possuem. A Figura 6.2 ilustra essa associação.

Como outro exemplo, considere os dois conjuntos de entidades *empréstimo* e *agência*. Podemos definir o conjunto de relacionamento *agência_empréstimo* para indicar a associação entre um empréstimo bancário e a agência em que esse empréstimo é mantido.

A associação entre conjuntos de entidades é chamada de **participação**; ou seja, os conjuntos de entidades $E_1, E_2, ..., E_n$ participam no conjunto de relacionamento R. Uma **instância de relacionamento** em um esquema E-R representa uma associação entre as entidades nomeadas nas empresas reais que estão sendo modeladas. Como uma ilustração, a entidade *cliente* individual Hayes, que possui o *id_cliente* 677-89-9011, e a entidade *empréstimo* L-15 participam em uma instância de relacionamento de *tomador*. Essa instância de relacionamento representa que, na empresa real, a pessoa chamada Hayes, que possui o *id_cliente* 677-89-9011, fez o empréstimo de número L-15.

A função desempenhada por uma entidade em um relacionamento é chamada de **papel** da entidade. Como os conjuntos de entidades participando em um conjunto de relacionamento geralmente são distintos, os papéis estão explícitos e normalmente não são especificados. Entretanto, eles são úteis quando a nomeação de um relacionamento precisa de esclarecimento. É o caso quando os conjuntos de entidades de um conjunto de relacionamento não são distintos; ou seja, o mesmo conjunto de entidades participa em um conjunto de relacionamento mais uma vez, em diferentes papéis. Nesse tipo de conjunto de relacionamento, às vezes chamado conjunto de relacionamento **recursivo**, são necessários nomes de papel explícitos para especificar como uma entidade participa em uma instância de relacionamento. Por exemplo, considere um conjunto de entidades *funcionário* que registra informações sobre todos os funcionários do banco. Podemos ter um conjunto de relacionamento *trabalha_para* que é modelado por pares ordenados de entida-

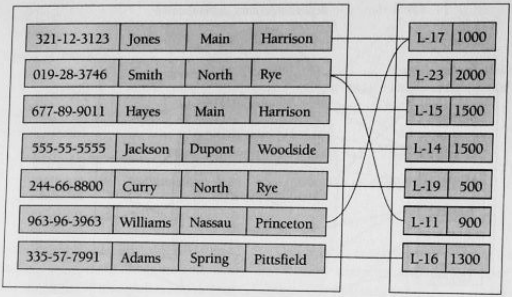


Figura 6.2 Conjunto de relacionamento *tomador*.

des *funcionário*. O primeiro funcionário de um par assume o papel de *trabalhador*, enquanto o segundo assume o papel de *gerente*. Dessa forma, todos os relacionamentos de *trabalha_para* são caracterizados por pares (trabalhador, gerente); os pares (gerente, trabalhador) são excluídos.

Um relacionamento também pode ter atributos chamados **atributos descritivos**. Considere um conjunto de relacionamento *depositante* com conjuntos de entidades *cliente* e *conta*. Podemos associar o atributo *data_acesso* a esse relacionamento para especificar a data mais recente em que um cliente acessou uma conta. O relacionamento *depositante* entre as entidades correspondentes ao cliente Jones e a conta A-217 tem o valor "23/Maio/2001".

A Figura 6.3 mostra o conjunto de relacionamento *depositante* com um atributo descritivo *data_acesso*; para manter a figura simples, apenas alguns atributos dos dois conjuntos de entidades são mostrados.

Como outro exemplo de atributos descritivos para relacionamentos, suponha que tenhamos conjuntos de entidades *aluno* e *curso*, que participam em um conjunto de relacionamento *registrado_para*. Podemos querer armazenar um atributo descritivo *para_crédito* com o relacionamento, para registrar se um aluno fez o curso para crédito ou o está freqüentando como ouvinte.

Uma instância de relacionamento em um determinado conjunto de relacionamento precisa ser identificável exclusivamente a partir de suas entidades participantes, sem usar os atributos descritivos. Para entender essa questão, suponha que queiramos modelar todos os dados de quando um cliente acessou uma conta. O atributo de valor único *data_acesso* só pode armazenar uma única data de acesso. Não podemos representar várias datas de acesso por múltiplas instâncias de relacionamento entre o mesmo cliente e conta, já que as instâncias de relacionamento não seriam identificáveis unicamente usando apenas as entidades participantes. A maneira correta de manipular esse caso é criar um atributo de múltiplos valores *datas_acesso*, que possa armazenar todas as datas de acesso.

Contudo, pode haver mais de um conjunto de relacionamento envolvendo os mesmos conjuntos de entidades. Em nosso exemplo, os conjuntos de entidades *cliente* e *empréstimo* participam no conjunto de relacionamento *tomador*. Suponha também que cada empréstimo precise ter outro cliente que sirva como fiador do empréstimo. Então, os conjuntos de entidades *cliente* e *empréstimo* podem participar em outro conjunto de relacionamento, *fiador*.

Os conjuntos de relacionamento *tomador* e *agência_empréstimo* fornecem um exemplo de um conjunto de relacionamento **binário** – ou seja, que envolve dois conjuntos de entidades. A maioria dos conjuntos de relacionamento em um sistema de banco de dados é binária. Ocasionalmente, entretanto, os conjuntos de relacionamento envolvem mais de dois conjuntos de entidades.

Como exemplo, considere os conjuntos de entidades *funcionário*, *agência* e *cargo*. Exemplos de entidades de *cargo* podem ser gerente, caixa, auditor etc. As entidades de *cargo* podem ter os atributos *título* e *nível*. O conjunto de relacionamento *trabalha_em* entre *funcionário*, *agência* e *cargo* é um exemplo de um relacionamento **ternário**. Um relacionamento ternário entre Jones, Perryridge e gerente indica que Jones atua como gerente na agência Perryridge. Jones também poderia atuar como auditor na agência

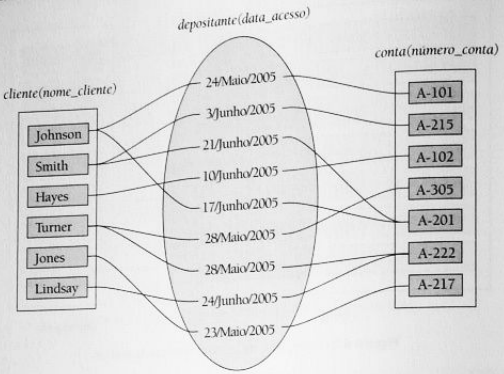


Figura 6.3 *data_acesso* como atributo do conjunto de relacionamento ***depositante***.

Downtown, que poderia ser representada por outro relacionamento. Outro relacionamento ainda poderia ser entre Smith, Downtown e caixa, indicando que Smith atua como caixa na agência Downtown.

O número de conjuntos de entidades que participam em um conjunto de relacionamento também é o **grau** do conjunto de relacionamento. Um conjunto de relacionamento binário possui o grau 2; um conjunto de relacionamento ternário possui o grau 3.

Atributos

Para cada atributo, existe um conjunto de valores permitidos, chamado o **domínio**, ou o **conjunto de valores**, desse atributo. O domínio do atributo *nome_cliente* poderia ser o conjunto de todas as strings de texto de um certo tamanho. Da mesma forma, o domínio do atributo *número_empréstimo* poderia ser o conjunto de todas as strings da forma "L-*n*", onde *n* é um inteiro positivo.

Formalmente, um atributo de um conjunto de entidades é uma função que mapeia do conjunto de entidades para um domínio. Como um conjunto de entidades pode ter vários atributos, cada entidade pode ser descrita por um conjunto de pares (atributo, valor de dados), um par para cada atributo do conjunto de entidades. Por exemplo, uma entidade *cliente* específica pode ser descrita pelo conjunto {(*id_cliente*, 677-89-9011), (*nome_cliente*, Hayes), (*rua_cliente*, Main), (*cidade_cliente*, Harrison)}, significando que a entidade descreve uma pessoa chamada Hayes cujo identificador de cliente é 677-89-9011 e que reside na Main Street, na cidade de Harrison. Podemos ver, a essa altura, uma integração entre o esquema abstrato e a empresa real sendo modelada. Os valores de atributo descrevendo uma entidade constituirão uma parte significativa dos dados armazenados no banco de dados.

Um atributo, como usado no modelo E-R, pode ser caracterizado pelos seguintes tipos de atributo:

- Atributos **simples** e **compostos**. Em nossos exemplos até agora, os atributos foram simples; ou seja, eles não foram divididos em subpartes. Os atributos compostos, por outro lado, podem ser divididos em subpartes (ou seja, em outros atributos). Por exemplo, um atributo *nome* poderia ser estruturado como um atributo composto consistindo em *prenome*, *inicial_meio* e *sobrenome*. Usar atributos compostos em um esquema de projeto é uma boa escolha se um usuário desejar se referir a um atributo inteiro em algumas ocasiões e a apenas um componente do atributo em outras ocasiões. Suponha que tivéssemos de substituir pelos atributos do conjunto de entidades *rua_cliente* e *cidade_cliente* de *cliente* o atributo composto *endereço* com os atributos *rua*, *cidade*, *estado* e *código_postal*.[2] Os atributos compostos nos ajudam a agrupar atributos relacionados, tornando a modelagem mais clara.

[2] Consideramos o formato de endereço usado nos Estados Unidos, que inclui um código postal numérico, chamado código postal.

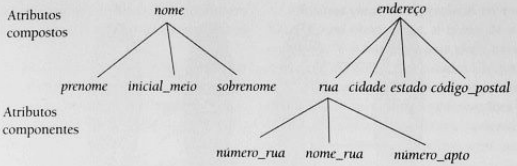


Figura 6.4 Atributos compostos *nome_cliente* e *endereço_cliente*.

Observe também que um atributo composto pode aparecer como uma hierarquia. No atributo composto *endereço*, seu atributo componente *rua* pode ser subdividido em *número_rua*, *nome_rua* e *número_apto*. A Figura 6.4 ilustra esses exemplos de atributos compostos para o conjunto de entidades *cliente*.

- Atributos de **valor único** e de **múltiplos valores**. Todos os atributos em nossos exemplos possuem um único valor para uma determinada entidade. Por exemplo, o atributo *número_empréstimo* para uma entidade *empréstimo* especifica se refere apenas a um número de empréstimo. Esses atributos são considerados como de **valor único**. Pode haver instâncias em que um atributo possui um conjunto de valores para uma entidade específica. Considere um conjunto de entidades *funcionário* com o atributo *número_telefone*. Um funcionário pode ter zero, um ou vários números de telefone, e diferentes funcionários podem ter diferentes números de telefone. Dizemos que esse tipo de atributo é de **múltiplos valores**. Como outro exemplo, um atributo *nome_dependente* do conjunto de entidades *funcionário* seria de múltiplos valores, já que qualquer funcionário específico pode ter zero, um ou mais dependentes.

  Quando apropriado, podem ser impostos limites superior e inferior no número de valores em um atributo de múltiplos valores. Por exemplo, um banco pode limitar a dois o número de telefones registrados para um único cliente. Colocar limites, nesse caso, expressa que o atributo *número_telefone* do conjunto de entidades *cliente* possa ter entre zero e dois valores.
- Atributo **derivado**. O valor para esse tipo de atributo pode ser derivado dos valores de outros atributos ou entidades relacionados. Por exemplo, digamos que o conjunto de entidades *cliente* possui um atributo *empréstimos_mantidos*, que representa a quantidade de empréstimos que um cliente tem do banco. Podemos derivar o valor para esse atributo contando o número de entidades *empréstimo* associadas a esse cliente.

  Como outro exemplo, suponha que o conjunto de entidades *cliente* tenha um atributo *idade* que indica a idade do cliente. Se o conjunto de entidades *cliente* também tiver um atributo *data_nascimento*, podemos calcular *idade* por meio de *data_nascimento* e da data atual. Portanto, *idade* é um atributo derivado. Nesse caso, *data_nascimento* pode ser referenciado como um atributo *base*, ou um atributo *armazenado*. O valor de um atributo derivado não é armazenado, mas é calculado quando necessário.

  Um atributo toma um valor **nulo** quando uma entidade não possui um valor para ele. O valor *nulo* pode indicar "não aplicável" – ou seja, que o valor não existe para a entidade. Por exemplo, alguém pode não ter um nome do meio. *Nulo* pode designar que um valor de atributo é desconhecido. Um valor desconhecido pode estar *omisso* (o valor existe, mas não temos essa informação) ou *desconhecido* (não sabemos se o valor realmente existe ou não).

  Por exemplo, se o valor *nome* para um cliente específico for *nulo*, consideramos que o valor está omisso, já que todo cliente precisa ter um nome. Um valor nulo para o atributo *número_apto* pode significar que o endereço não inclui um número de apartamento (não aplicável), que um número de apartamento existe mas não sabemos qual ele é (omisso) ou que não sabemos se um número de apartamento é parte do endereço do cliente (desconhecido).

Restrições

Em esquema de empresa E-R pode definir certas restrições às quais o conteúdo de um banco de dados precisa se conformar. Nesta seção, examinaremos as cardinalidades de mapeamento, as restrições de chave e as restrições de participação.

Cardinalidades de mapeamento

As **cardinalidades de mapeamento**, ou fatores de cardinalidade, expressam o número de entidades ao qual outra entidade pode ser associada por um conjunto de relacionamento.

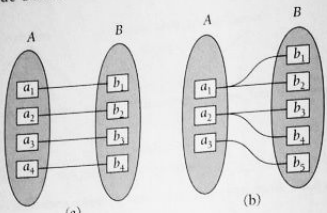


Figura 6.5 Cardinalidades de mapeamento. (a) Um-para-um. (b) Um-para-muitos.

As cardinalidades de mapeamento são úteis principalmente em descrever conjuntos de relacionamento binários, embora possam contribuir para a descrição dos conjuntos de relacionamento que envolvem mais de dois conjuntos de entidades. Nesta seção, nos concentraremos apenas nos conjuntos de relacionamento binários.

Para um conjunto de relacionamento binário R entre conjuntos de entidades A e B, a cardinalidade de mapeamento precisa ser uma das seguintes:

- **Um-para-um**. Uma entidade em A é associada a *no máximo* uma entidade em B, e uma entidade em B é associada a *no máximo* uma entidade em A. (Veja a Figura 6.5a.)
- **Um-para-muitos**. Uma entidade em A é associada a qualquer número de entidades (zero ou mais) em B. Entretanto, uma entidade em B pode ser associada a *no máximo* uma entidade em A. (Veja a Figura 6.5b.)
- **Muitos-para-um**. Uma entidade em A é associada a *no máximo* uma entidade em B. Entretanto, uma entidade em B pode ser associada a qualquer número de entidades (zero ou mais) em A. (Veja a Figura 6.6a.)
- **Muitos-para-muitos**. Uma entidade em A é associada a qualquer número de entidades (zero ou mais) em B, e uma entidade em B pode ser associada a qualquer número de entidades (zero ou mais) em A. (Veja a Figura 6.6b.)

A cardinalidade de mapeamento apropriada para um determinado conjunto de relacionamento, obviamente, depende da situação real que o conjunto de relacionamento está modelando.

Como ilustração, considere o conjunto de relacionamento *tomador*. Se, em um determinado banco, um empréstimo pode pertencer a apenas um cliente, e um cliente pode ter vários empréstimos, então, o conjunto de relacionamento de *cliente* para *empréstimo* é um-para-muitos. Se um empréstimo pode pertencer a vários clientes (como os empréstimos tomados conjuntamente por vários sócios), o conjunto de relacionamento é muitos-para-muitos. A Figura 6.2 ilustra esse tipo de relacionamento.

Chaves

Precisamos ter uma maneira de especificar como as entidades dentro de um determinado conjunto de entidades são distinguidas. Conceitualmente, as entidades individuais

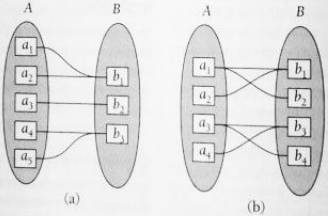


Figura 6.6 Cardinalidades de mapeamento. (a) Muitos-para-um. (b) Muitos-para-muitos.

são distintas; de uma perspectiva de banco de dados, no entanto, a diferença entre elas precisa ser expressa em termos de seus atributos.

Portanto, os valores atributos de uma entidade precisam ser tais que possam *identificar unicamente* a entidade. Em outras palavras, nenhuma entidade em um conjunto de entidades pode ter exatamente o mesmo valor de outra entidade para todos os atributos.

Uma *chave* permite identificar um conjunto dos atributos que são suficientes para distinguir uma entidade das outras. As chaves também ajudam a identificar relacionamentos unicamente e, assim, distinguir relacionamentos uns dos outros.

Conjuntos de entidades

Uma **superchave** é um conjunto de um ou mais atributos que, tomados coletivamente, permitem identificar unicamente uma entidade no conjunto de entidades. Por exemplo, o atributo *id_cliente* do conjunto de entidades *cliente* é suficiente para distinguir uma entidade *cliente* de outra. Portanto, *id_cliente* é uma superchave. Do mesmo modo, a combinação de *nome_cliente* e *id_cliente* é uma superchave para o conjunto de entidades *cliente*. O atributo *nome_cliente* de *cliente* não é uma superchave, pois várias pessoas podem ter o mesmo nome.

O conceito de uma superchave não é suficiente para nossos propósitos, já que, como vimos, uma superchave pode conter atributos estranhos. Se K for uma superchave, também o será qualquer superconjunto de K. Muitas vezes, estamos interessados nas superchaves para as quais nenhum subconjunto apropriado é uma superchave. Essas superchaves mínimas são chamadas de **chaves candidatas**.

É possível que vários conjuntos distintos de atributos possam servir como chave candidata. Suponha que uma combinação de *nome_cliente* e *rua_cliente* seja suficiente para distinguir entre membros do conjunto de entidades *cliente*. Então, tanto {*id_cliente*} quanto {*nome_cliente*, *rua_cliente*} são chaves candidatas. Embora os atributos *id_cliente* e *nome_cliente* juntos possam distinguir entidades de *cliente*, sua combinação não forma uma chave candidata, já que o atributo *id_cliente* sozinho é uma chave candidata.

Usaremos o termo **chave primária** para indicar uma chave candidata que é escolhida pelo projetista de banco de dados como o principal meio de identificar entidades dentro de um conjunto de entidades. Uma chave (primária, candidata e super) é uma propriedade do conjunto de entidades, e não as entidades individuais. Quaisquer duas entidades individuais no conjunto são proibidas de ter o mesmo valor nos atributos de chave ao mesmo tempo. A designação de uma chave representa uma restrição na empresa real sendo modelada.

As chaves candidatas precisam ser escolhidas com cuidado. Como observamos, o nome de uma pessoa obviamente não é o bastante, pois pode haver muitas pessoas com o mesmo nome. Nos Estados Unidos, o atributo de número de seguro social de uma pessoa seria uma chave candidata. Como as pessoas residentes fora dos Estados Unidos normalmente não possuem números de seguro social, as empresas internacionais precisam gerar seus próprios identificadores únicos. Uma alternativa é usar alguma combinação única de outros atributos como chave.

A chave primária deve ser escolhida de modo que seus atributos nunca, ou raramente, sejam mudados. Por exemplo, o campo *endereço* de uma pessoa não deve ser parte da chave primária, já que ele provavelmente mudará. Os números de seguro social, por outro lado, têm a garantia de nunca mudar. Os identificadores únicos gerados pelas empresas normalmente não mudam, exceto se houver a fusão de duas empresas; nesse caso, o mesmo identificador pode ter sido emitido pelas duas empresas, e uma realocação de identificadores pode ser necessária para garantir que sejam únicos.

Conjuntos de relacionamento

A chave primária de um conjunto de entidades permite distinguir entre as várias entidades do conjunto. Precisamos de um mecanismo semelhante para distinguir entre os vários relacionamentos de um conjunto de relacionamento.

Considere R um conjunto de relacionamento envolvendo os conjuntos de entidades $E_1, E_2, \ldots, E_n$. Seja *chave primária*(E_i) o conjunto de atributos que forma a chave primária para o conjunto de entidades E_i. Considere por enquanto que os nomes de atributo de todas as chaves primárias são únicos e cada conjunto de entidades participa apenas uma vez no relacionamento. A composição da chave primária para um conjunto de relacionamento depende do conjunto de atributos associados ao conjunto de relacionamento R.

Se o conjunto de relacionamento R não tiver atributos associados a ele, então, o conjunto de atributos

$$\textit{chave primária}(E_1) \cup \textit{chave primária}(E_2) \cup \ldots \cup \textit{chave primária}(E_n)$$

descreve um relacionamento individual no conjunto R.

Se o conjunto de relacionamento R possui atributos $a_1, a_2, \ldots, a_m$ associados a ele, então, o conjunto de atributos

$$\textit{chave primária}(E_1) \cup \textit{chave primária}(E_2) \cup \ldots \cup \textit{chave primária}(E_n) \cup \{a_1, a_2, \ldots, a_m\}$$

descreve um relacionamento individual no conjunto R.

Nesses dois casos, o conjunto de atributos

chave primária(E_1) ∪ *chave primária*(E_2) ∪ ... ∪ *chave primária*(E_n)

forma uma superchave para o conjunto de relacionamento.

No caso de os nomes de atributo das chaves primárias não serem únicos entre os conjuntos de entidades, os atributos são renomeados para distingui-los; o nome do conjunto de entidades combinado com o nome do atributo formaria um nome único. No caso de um conjunto de entidades participar mais de uma vez em um conjunto de relacionamento (como no relacionamento *trabalha_para* da seção "Conjuntos de relacionamento"), o nome de papel é usado no lugar do nome do conjunto de entidades, para formar um nome de atributo único.

A estrutura da chave primária para o conjunto de relacionamento depende da cardinalidade de mapeamento do conjunto de relacionamento. Por exemplo, considere os conjuntos de entidades *cliente* e *conta*, e o conjunto de relacionamento *depositante*, com o atributo *data_acesso* da seção "Conjuntos de relacionamento". Suponha que o conjunto de relacionamento seja muitos-para-muitos. Então, a chave primária de *depositante* consiste na união das chaves primárias de *cliente* e *conta*. Entretanto, se um cliente puder ter apenas uma conta – isto é, se o relacionamento *depositante* for muitos-para-um de *cliente* para *conta* – então, a chave primária de *depositante* será simplesmente a chave primária de *cliente*. Da mesma forma, se o relacionamento for muitos-para-um de *conta* para *cliente* – isto é, cada conta pertence no máximo a um cliente – então, a chave primária de *depositante* será simplesmente a chave primária de *conta*. Para relacionamentos um-para-um, qualquer chave primária pode ser usada.

Para relacionamentos não binários, se nenhuma restrição de cardinalidade estiver presente, a superchave formada como descrito anteriormente nesta seção será a única chave candidata, portanto será escolhida como a chave primária.[1] A escolha da chave primária é mais complicada se restrições de cardinalidade estão presentes. Já que não abordamos como especificar restrições de cardinalidade em relações não binárias, não discutiremos mais essa questão neste capítulo e a consideraremos mais detalhadamente na seção "Teoria da dependência funcional" do Capítulo 7.

Restrições de participação

A participação de um conjunto de entidades E em um conjunto de relacionamento R é chamada de **total** se todas as entidades em E participam em pelo menos um relacionamento em R. Se apenas algumas entidades em E participam em relacionamentos em R, a participação do conjunto de entidades E no relacionamento R é chamada de **parcial**. Por exemplo, esperamos que toda entidade empréstimo esteja relacionada a pelo menos um cliente por meio do relacionamento *tomador*. Portanto, a participação de *empréstimo* no conjunto de relacionamento *tomador* é total. Por outro lado, uma pessoa pode ser um cliente de banco quer ela tenha ou não um empréstimo com o banco. Consequentemente, é possível que apenas algumas das entidades *cliente* estejam relacionadas ao conjunto de entidades *empréstimo* por meio do relacionamento *tomador*, e a participação de *cliente* no conjunto de relacionamento *tomador*, portanto, é parcial.

Diagramas de entidade-relacionamento

Como vimos brevemente na seção "Modelos de dados" do Capítulo 1, um **diagrama E-R** pode expressar graficamente a estrutura lógica geral de um banco de dados. Os diagramas E-R são simples e claros – qualidades que podem ter motivado o amplo uso do modelo E-R. Esse diagrama consiste nos seguintes componentes principais:

- **Retângulos**, que representam conjuntos de entidades
- **Elipses**, que representam atributos
- **Losangos**, que representam conjuntos de relacionamento
- **Linhas**, que vinculam atributos a conjuntos de entidades e estes a conjuntos de relacionamento
- **Elipses duplas**, que representam atributos de valores multiplos
- **Elipses tracejadas**, que representam atributos derivados
- **Linhas duplas**, que indicam participação total de uma entidade em um conjunto de relacionamento
- **Retângulos duplos**, que representam conjuntos de entidades fracos (descritos mais adiante, na seção "Conjuntos de entidades fracos")

Considere o diagrama de entidade-relacionamento na Figura 6.7, que consiste em dois conjuntos de entidades, *cliente* e *empréstimo*, relacionados por um conjunto de relacionamento binário *tomador*. Os atributos associados a *cliente* são *id_cliente*, *nome_cliente*, *rua_cliente* e *cidade_cliente*. Os atributos associados a *empréstimo* são *número_empréstimo* e *quantia*. Na Figura 6.7, os atributos de um conjunto de entidades que são membros da chave primária estão sublinhados.

O conjunto de relacionamento *tomador* pode ser muitos-para-muitos, um-para-muitos, muitos-para-um ou um-para-um. Para distinguir entre esses tipos, desenhamos uma linha direcionada (↦) ou uma linha não direcionada (–) entre o conjunto de relacionamento e o conjunto de entidades em questão.

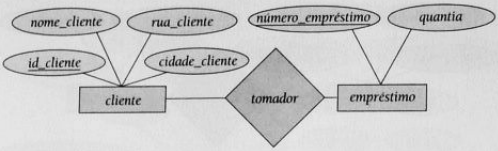


Figura 6.7 Diagrama E-R correspondente a clientes e empréstimos.

- Uma linha direcionada do conjunto de relacionamento *tomador* até o conjunto de entidades *empréstimo* especifica que *tomador* é um conjunto de relacionamento um-para-um ou muitos-para-um, de *cliente* para *empréstimo*; *tomador* não pode ser um conjunto de relacionamento muitos-para-muitos ou um-para-muitos de *cliente* para *empréstimo*.
- Uma linha não direcionada do conjunto de relacionamento *tomador* até o conjunto de entidades *empréstimo* especifica que *tomador* é um conjunto de relacionamento muitos-para-muitos ou um-para-muitos de *cliente* para *empréstimo*.

Voltando para o diagrama E-R da Figura 6.7, vemos que o conjunto de relacionamento *tomador* é muitos-para-muitos. Se o conjunto de relacionamento *tomador* fosse um-para-muitos, de *cliente* para *empréstimo*, então, a linha de *tomador* para *cliente* seria direcionada, com uma seta apontando para o conjunto de entidades *cliente* (Figura 6.8a). Da mesma forma, se o conjunto de relacionamento *tomador* fosse muitos-para-um de *cliente* para *empréstimo*, então, a linha de *tomador* para *empréstimo* teria uma seta apontando para o conjunto de entidades *empréstimo* (Figura 6.8b). Finalmente, se o conjunto de relacionamento *tomador* fosse um-para-um, as duas linhas de *tomador* teriam setas: uma apontando para o conjunto de entidades *empréstimo* e uma apontando para o conjunto de entidades *cliente* (Figura 6.8c).

Se um conjunto de relacionamento também tiver alguns atributos associados a ele, então, vinculamos esses atributos a esse conjunto de relacionamento. Por exemplo, na Figura 6.9, temos o atributo descritivo *dados_acesso* conectado ao conjunto de relacionamento *depositante* para especificar a data mais recente em que um cliente acessou essa conta.

A Figura 6.10 mostra como atributos compostos podem ser representados na notação E-R. Aqui, um atributo composto *nome*, com atributos componentes *prenome*, *inicial_meio* e *sobrenome*, substitui o atributo simples *nome_cliente* de *cliente*. Além disso, um atributo composto *endereço*, cujos atributos componentes são *rua*, *cidade*, *estado* e *código_postal*, substitui os atributos *rua_cliente* e *cidade_cliente* de *cliente*. O atributo *rua* é, ele próprio, um atributo composto cujos atributos componentes são *número_rua*, *nome_rua* e *número_apto*.

A Figura 6.10 também ilustra um atributo com valores múltiplos *número_telefone*, representado por uma elipse dupla, e um atributo derivado *idade*, representado por uma elipse tracejada.

Indicamos papéis nos diagramas E-R rotulando as linhas que conectam losangos a retângulos. A Figura 6.11 mostra os indicadores de papel *gerente* e *trabalhador* entre o conjunto de entidades *funcionário* e o conjunto de relacionamento *trabalha_para*.

Os conjuntos de relacionamento binários podem ser especificados facilmente em um diagrama E-R. A Figura 6.12 consiste nos três conjuntos de entidades, *funcionário*, *cargo* e *agência*, relacionados pelo conjunto de relacionamento *trabalha_em*.

Podemos especificar alguns tipos de relacionamentos muitos-para-um no caso dos conjuntos de relacionamento binários. Suponha que um funcionário pode ter no máximo um cargo em cada agência (por exemplo, Jones não pode ser um gerente e um auditor da mesma agência). Essa restrição pode ser especificada por uma seta apontando para *cargo* na borda de *trabalha_em*.

Permitimos no máximo uma seta de um conjunto de relacionamento, já que um diagrama E-R com duas ou mais setas de um conjunto de relacionamento não binário pode ser interpretado de duas maneiras. Suponha que exista um conjunto de relacionamento R entre os conjuntos de entidades $A_1, A_2, \ldots, A_n$ e as únicas setas estão nas bordas para os conjuntos de entidades $A_{i+1}, A_{i+2}, \ldots, A_n$. Portanto, as duas interpretações possíveis são:

1. Uma determinada combinação de entidades de $A_1, A_2, \ldots, A_i$ pode estar associada a no máximo uma combinação de entidades de $A_{i+1}, A_{i+2}, \ldots, A_n$. Portanto, a chave primária para o relacionamento R pode ser construída pela união das chaves primárias de $A_1, A_2, \ldots, A_i$.

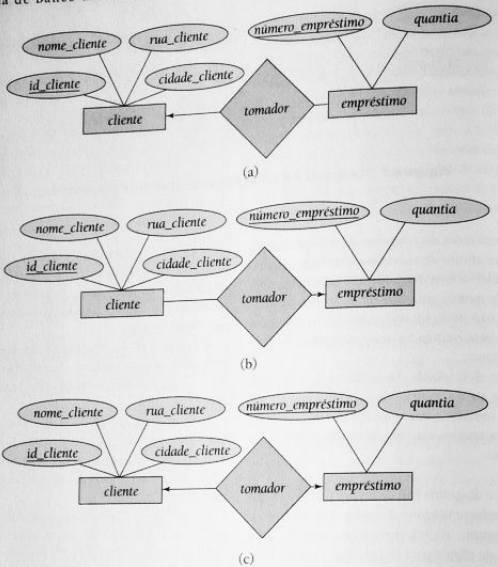


Figura 6.8 Relacionamentos. (a) Um-para-muitos. (b) Muitos-para-um. (c) Um-para-um.

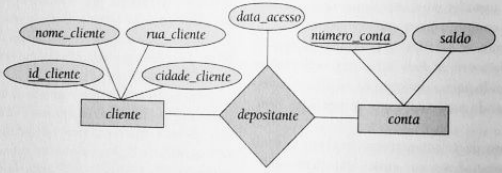


Figura 6.9 Diagrama E-R com um atributo conectado a um conjunto de relacionamento.

2. Para cada conjunto de entidades A_k, $i < k \leq n$, cada combinação de entidades dos outros conjuntos de entidades pode estar associada a no máximo uma entidade de A_k. Cada conjunto $\{A_1, A_2, \ldots, A_{k-1}, A_{k+1}, \ldots, A_n\}$, para $i < k \leq n$, então, forma uma chave candidata.

Cada uma dessas interpretações tem sido usada em diferentes livros e sistemas. Para evitar confusão, permitimos apenas uma seta de um conjunto de relacionamento, caso em que as duas interpretações são equivalentes. No Capítulo 7 (seção "Teoria da dependência funcional") estudamos a noção de *dependências funcionais*, que permite que qual-

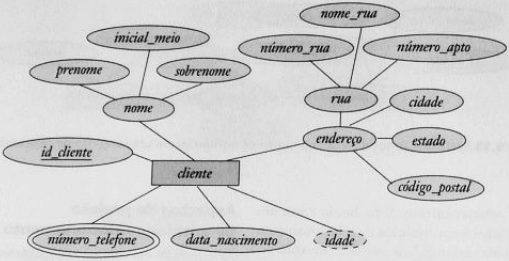


Figura 6.10 Diagrama E-R com atributos compostos, com valores múltiplos e derivados.

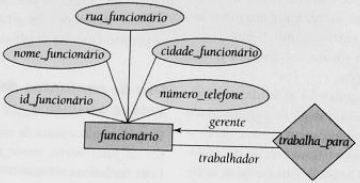


Figura 6.11 Diagrama E-R com indicadores de papel.

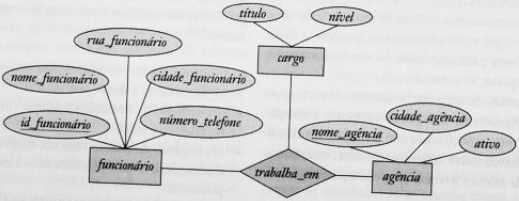


Figura 6.12 Diagrama E-R com um relacionamento ternário.

quer uma dessas interpretações seja especificada de uma maneira não ambígua.

Linhas duplas são usadas em um diagrama E-R para indicar que a participação de um conjunto de entidades em um conjunto de relacionamento é total; ou seja, cada entidade no conjunto de entidades ocorre em pelo menos um relacionamento nesse conjunto de relacionamento. Por exemplo, considere o relacionamento *tomador* entre clientes e empréstimos. Uma linha dupla de *empréstimo* a *tomador*, como na Figura 6.13, indica que cada empréstimo precisa ter pelo menos um cliente associado.

Os diagramas E-R também fornecem uma maneira de indicar restrições mais complexas sobre o número de vezes que cada entidade participa em relacionamentos em

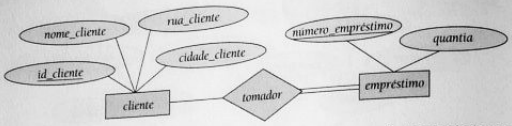


Figura 6.13 Participação total de um conjunto de entidades em um conjunto de relacionamento.

um conjunto de relacionamento. Uma borda entre um conjunto de entidades e um conjunto de relacionamento binário pode ter uma cardinalidade mínima e máxima associada, mostrada na forma *l..h*, onde *l* é a cardinalidade mínima e *h*, a cardinalidade máxima. Um valor mínimo de 1 indica participação total do conjunto de entidades no conjunto de relacionamento. Um valor máximo de 1 indica que a entidade participa no máximo em um relacionamento, enquanto um valor máximo * indica nenhum limite. Observe que um rótulo 1..* em uma borda é equivalente a uma linha dupla.

Por exemplo, considere a Figura 6.14. A borda entre *empréstimo* e *tomador* possui uma restrição de cardinalidade de 1..1, significando que a cardinalidade mínima e máxima são, ambas, 1. Ou seja, cada empréstimo precisa ter exatamente um cliente associado. O limite 0..* na borda de *cliente* para *tomador* indica que um cliente pode ter zero ou mais empréstimos. Portanto, o relacionamento *tomador* é um-para-muitos de *cliente* para *empréstimo*, e a participação de *empréstimo* em *tomador* é total.

É fácil interpretar errado o 0..* na borda entre *cliente* e *tomador*, e pensar que o relacionamento *tomador* é muitos-para-um de *cliente* para *empréstimo* – isso é exatamente o inverso da interpretação correta.

Se ambas as bordas de um relacionamento binário possuem um valor máximo de 1, o relacionamento é um-para-um. Se tivéssemos especificado um limite de cardinalidade de 1..* na borda entre *cliente* e *tomador*, estaríamos dizendo que cada cliente precisa ter pelo menos um empréstimo.

Aspectos de projeto de entidade-relacionamento

As noções de um conjunto de entidades e um conjunto de relacionamento não são precisas, e é possível definir um conjunto de entidades e os relacionamentos entre eles de diversas maneiras. Nesta seção, examinaremos os aspectos básicos no projeto de um esquema de banco de dados E-R. A seção "Restrições em generalizações" aborda o processo de projeto em mais detalhes.

Uso de conjuntos de entidades versus atributos

Considere o conjunto de entidades *funcionário* com atributos *id_funcionário*, *nome_funcionário* e *número_telefone*. Pode ser facilmente questionado que um telefone é uma entidade em seu próprio direito, com atributos *número_telefone* e *local*; o local pode ser o escritório ou casa onde o telefone está instalado, com telefones celulares talvez representados pelo valor "móvel". Se tomarmos esse ponto de vista, precisaremos redefinir o conjunto de entidades *funcionário* como:

- O conjunto de entidades *funcionário*, com atributos *id_funcionário* e *nome_funcionário*
- O conjunto de entidades *telefone*, com atributos *número_telefone* e *local*
- O conjunto de relacionamento *telefone_func*, indicando a associação entre os funcionários e os telefones que eles possuem

Essas alternativas são mostradas na Figura 6.15.

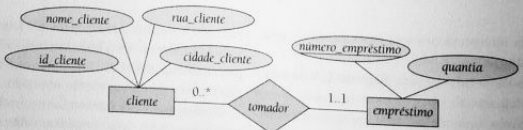
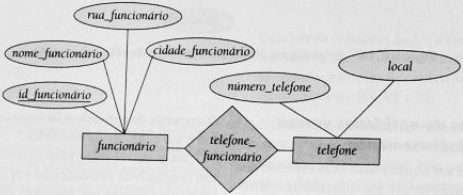


Figura 6.14 Limites de cardinalidade em conjuntos de relacionamento.

(a)



(b)

Figura 6.15 Alternativas para *funcionário* e *telefone*.

Então, qual é a principal diferença entre essas duas definições de um funcionário? Tratar um telefone como um atributo *número_telefone* implica que cada funcionário tem precisamente um número de telefone. Tratar um telefone como uma entidade *telefone* permite que os funcionários tenham vários números de telefone (incluindo zero) associados a eles. Entretanto, poderíamos, em vez disso, definir facilmente *número_telefone* como um atributo de valores multiplos para permitir vários telefones por funcionário.

A principal diferença, então, é que tratar um telefone como uma entidade modela melhor uma situação em que se pode querer manter informações extras sobre um telefone, como seu local ou seu tipo (móvel, videofone ou telefone convencional), ou todas as pessoas que compartilham o telefone. Portanto, tratar um telefone como uma entidade é mais geral que tratá-lo como um atributo e é apropriado quando a generalidade pode ser útil.

Por outro lado, não seria apropriado tratar o atributo *nome_funcionário* como uma entidade; é difícil questionar que *nome_funcionário* seja uma entidade em seu próprio direito (ao contrário do telefone). Assim, é apropriado ter *nome_funcionário* como um atributo do conjunto de entidades *funcionário*.

Duas perguntas naturais surgem então: o que constitui um atributo e o que constitui um conjunto de entidades? Infelizmente, não existem respostas simples. As diferenças dependem principalmente da estrutura da empresa real sendo modelada e da semântica associada com o atributo em questão.

Um erro comum é usar a chave primária de um conjunto de entidades como um atributo de outro conjunto de entidades, em vez de usar um relacionamento. Por exemplo, não é correto modelar *id_cliente* como um atributo de *empréstimo*, mesmo se cada empréstimo tivesse apenas um cliente. O relacionamento *tomador* é a maneira correta de representar a conexão entre empréstimos e clientes, já que ele torna sua conexão explícita, e não implícita, por meio de um atributo.

Outro erro relacionado que as pessoas algumas vezes cometem é designar os atributos de chave primária dos conjuntos de entidades relacionados como atributos do conjunto de relacionamento. Por exemplo, *número_empréstimo* (o atributo de chave primária de *empréstimo*) e *id_cliente* (a chave primária de *cliente*) não devem aparecer como atributos do relacionamento *tomador*. Isso não deveria ser feito, já que os atributos de chave primária já estão implícitos no conjunto de relacionamento.[3]

3. Quando criamos um esquema de relacionamento a partir do esquema E-R, os atributos podem aparecer em uma tabela criada do conjunto de relacionamento *tomador*, como veremos mais adiante; entretanto, eles não devem aparecer no conjunto de relacionamento *tomador*.

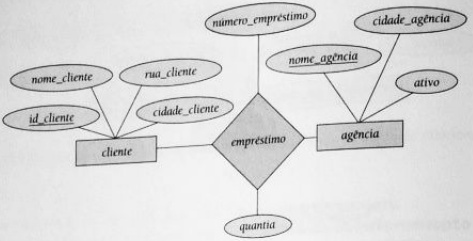


Figura 6.16 *Empréstimo* como um conjunto de relacionamento.

Uso de conjuntos de entidades versus conjuntos de relacionamento

Nem sempre é claro se um objeto é mais bem expresso por um conjunto de entidades ou por um conjunto de relacionamento. Na seção "Conjuntos de entidades", consideramos que um empréstimo bancário é modelado como uma entidade. Uma alternativa é modelar um empréstimo não como uma entidade, mas como um relacionamento entre clientes e agências, com *número_empréstimo* e *quantia* como atributos descritivos, como mostra a Figura 6.16. Cada empréstimo é representado por um relacionamento entre um cliente e uma agência.

Se cada empréstimo é mantido exatamente por um cliente e está associado exatamente a uma agência, podemos achar satisfatório o projeto em que um empréstimo é representado como um relacionamento. Entretanto, com esse projeto, não podemos representar de modo conveniente uma situação em que vários clientes mantêm um empréstimo conjuntamente. Para tratar dessa situação, precisamos definir um relacionamento separado para cada mantenedor do empréstimo conjunto. Então, precisamos duplicar os valores para os atributos descritivos número_empréstimo e quantia em cada um desses relacionamentos. Cada um deles, é claro, precisa ter o mesmo valor para os atributos descritivos número_empréstimo e quantia.

Dois problemas surgem como resultado da duplicação: (1) os dados são armazenados várias vezes, desperdiçando espaço de armazenamento, e (2) as atualizações podem deixar os dados em um estado inconsistente, em que os valores diferem em dois relacionamentos para atributos que devem ter o mesmo valor. O problema de como evitar essa duplicação é tratado formalmente pela teoria da normalização, abordada no Capítulo 7.

O problema da duplicação dos atributos número_empréstimo e quantia está ausente no projeto original da seção "Diagramas de entidade-relacionamento", pois lá empréstimo é um conjunto de entidades.

Uma possível regra para determinar se é melhor usar um conjunto de entidades ou um conjunto de relacionamento é designar um conjunto de relacionamento para descrever uma ação que ocorre entre entidades. Essa técnica também pode ser útil para decidir se certos atributos podem ser expressos mais apropriadamente como relacionamentos.

Conjuntos de relacionamento binários versus enários

Os relacionamentos nos bancos de dados normalmente são binários. Alguns relacionamentos que parecem ser não binários podem, na verdade, ser mais bem representados por vários relacionamentos binários. Por exemplo, seria possível criar um relacionamento ternário pais, relacionando um filho ao seu pai e mãe. Entretanto, esse relacionamento também poderia ser representado por dois relacionamentos binários, mãe e pai, relacionando um filho à sua mãe e ao seu pai separadamente. Usar os dois relacionamentos, mãe e pai, fornece um registro da mãe de um filho, mesmo se não conhecermos a identidade do pai; um valor nulo seria necessário se o relacionamento ternário pais fosse usado. Nesse caso, é preferível usar conjuntos de relacionamento binários.

Na realidade, é sempre possível substituir um conjunto de relacionamento não binário (enário, para $n > 2$) por diversos conjuntos de relacionamento binários distintos. Para simplicidade, considere o conjunto de relacionamento ternário abstrato ($n = 3$) R, relacionando conjuntos de entidades A, B e C. Substituímos o conjunto de relacionamento R por um conjunto de entidades E e criamos três conjuntos de relacionamento, como mostra a Figura 6.17:

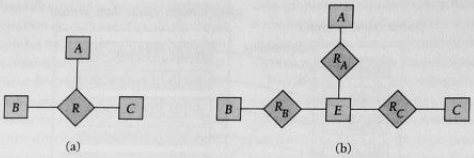


Figura 6.17 Relacionamento ternário comparado com três relacionamentos binários.

- R_A, relacionando E e A
- R_B, relacionando E e B
- R_C, relacionando E e C

Se o conjunto de relacionamento R tivesse quaisquer atributos, esses seriam atribuídos ao conjunto de entidades E; além disso, um atributo de identificação especial é criado para E (já deve ser possível distinguir diferentes entidades em um conjunto de entidades na base de seus valores de atributo). Para cada relacionamento (ai, bi, ci) no conjunto de relacionamento R, criamos uma nova entidade ei no conjunto de entidades E. Depois, em cada um dos três novos conjuntos de relacionamento, inserimos um relacionamento desta forma:

- (e_i, a_i) em R_A
- (e_i, b_i) em R_B
- (e_i, c_i) em R_C

Podemos generalizar esse processo de uma maneira simples para conjuntos de relacionamento enários. Assim, conceitualmente, podemos restringir o modelo E-R para incluir apenas conjuntos de relacionamento binários. Entretanto, essa restrição nem sempre é desejável.

- Um atributo identificador pode ter sido criado para o conjunto de entidades criado a fim de representar o conjunto de relacionamento. Esse atributo, juntamente com os conjuntos de relacionamento extras, aumenta a complexidade do projeto e (como veremos na seção "Redução aos esquemas relacionais") as necessidades de armazenamento gerais.
- Um conjunto de relacionamento enário mostra mais claramente que várias entidades participam em um único relacionamento.
- Pode não haver um meio de traduzir restrições no relacionamento ternário para restrições nos relacionamentos binários. Por exemplo, considere uma restrição que diga que R é muitos-para-um de A, B para C; ou seja, cada par de entidades de A e B está associado a no máximo uma entidade C. Essa restrição não pode ser expressa usando restrições de cardinalidade nos conjuntos de relacionamento RA, RB e RC.

Considere o conjunto de relacionamento trabalha_em na seção "Conjuntos de relacionamento", relacionando *funcionário*, *agência* e *cargo*. Não podemos dividir diretamente *trabalha_em* em relacionamentos binários entre *funcionário* e *agência* e entre *funcionário* e *cargo*. Se fizéssemos isso, seríamos capazes de registrar que Jones é gerente e auditor e que Jones trabalha nas agências Perryridge e Downtonwn. Entretanto, não poderíamos registrar que Jones é gerente em Perryridge e auditor em Downtown, mas não auditor em Perryridge ou gerente em Downtown.

O conjunto de relacionamento *trabalha_em* pode ser dividido em relacionamentos binários criando um novo conjunto de entidades, conforme descrito anteriormente. No entanto, fazer isso não seria muito natural.

Posicionamento dos atributos de relacionamento

A razão de cardinalidade de um relacionamento pode afetar o posicionamento dos atributos de relacionamento. Portanto, os atributos dos relacionamentos um-para-um ou um-para-muitos podem estar associados a um dos conjuntos de entidades participantes, e não ao conjunto de relacionamento. Por exemplo, vamos especificar que *depositante* é um conjunto de relacionamento um-para-muitos tal que um cliente pode ter várias contas, mas cada conta é mantida apenas por um cliente. Nesse caso, o atributo *data_acesso*, que especifica quando o cliente acessou pela última vez essa conta, poderia ser associado ao conjunto de entidades *conta*, como ilustra a Figura 6.18; para manter a figura simples, apenas alguns dos atributos dos dois conjuntos de entidades são mostrados. Como cada entidade *conta* participa em um relacionamento com no máximo uma instância de *cliente*, criar essa designação de atributo teria o mesmo significado de colocar *data_acesso* com o

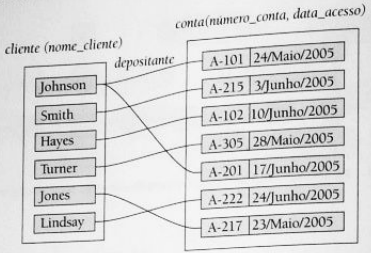


Figura 6.18 *data_acesso* como atributo do conjunto de entidades *conta*.

conjunto de relacionamento *depositante*. Os atributos de um conjunto de relacionamento um-para-muitos podem ser reposicionados apenas para o conjunto de entidades no lado "muitos" do relacionamento. Entretanto, para conjuntos de relacionamento um-para-um, o atributo de relacionamento pode ser associado a qualquer uma das entidades participantes.

A decisão de projeto de onde colocar atributos descritivos nesses casos – como um atributo de relacionamento ou entidade – deve refletir as características da empresa sendo modelada. O projetista pode escolher manter *data_acesso* como um atributo de *depositante* para expressar explicitamente que um acesso ocorre no ponto de interação entre os conjuntos de entidades *cliente* e *conta*.

A escolha do posicionamento de atributo é mais simples para conjuntos de relacionamento muitos-para-muitos. Voltando ao nosso exemplo, vamos especificar o caso talvez mais realista de que *depositante* é um conjunto de relacionamento muitos-para-muitos, expressando que um cliente pode ter uma ou mais contas, e que uma conta pode ser mantida por um ou mais clientes. Se formos expressar a data em que um determinado cliente acessou pela última vez uma conta específica, *data_acesso* precisará ser um atributo do relacionamento *depositante*, em vez de qualquer uma das entidades participantes. Se *data_acesso* fosse um atributo de *conta*, por exemplo, não poderíamos determinar que cliente fez o acesso mais recente a uma conta conjunta. Quando um atributo é determinado pela combinação dos conjuntos de entidades participantes, e não por qualquer entidade separadamente, esse atributo precisa ser associado ao relacionamento muitos-para-muitos. A Figura 6.3 mostra o posicionamento de *data_acesso* como um atributo de relacionamento; novamente, para manter a figura simples, apenas alguns dos atributos dos dois conjuntos de entidades são mostrados.

Conjuntos de entidades fracos

Um conjunto de entidades pode não ter atributos suficientes para formar uma chave primária. Esse conjunto de entidades é denominado **conjunto de entidades fraco**. Um conjunto de entidades que possui uma chave primária é chamado **conjunto de entidades forte**.

Como ilustração, considere o conjunto de entidades *pagamento*, que tem três atributos: *número_pagamento*, *data_pagamento* e *quantia_pagamento*. Os números de pagamento normalmente são números seqüenciais, começando com 1, gerados separadamente para cada empréstimo. Portanto, embora cada entidade *pagamento* seja distinta, os pagamentos para diferentes empréstimos podem compartilhar o mesmo número de pagamento. Assim, esse conjunto de entidades não possui uma chave primária; ele é um conjunto de entidades fraco.

Para um conjunto de entidades fraco ser significativo, ele precisa estar associado a outro conjunto de entidades, chamado o **conjunto de entidades identificador**, ou **proprietário**. Toda entidade fraca precisa estar associada a uma entidade identificadora; ou seja, dizemos que o conjunto de entidades fraco é **dependente de existência** do conjunto de entidades identificador. Dizemos que o conjunto de entidades identificador **possui** o conjunto de entidades fraco que ele identifica. O relacionamento associando o conjunto de entidades fraco com o conjunto de entidades identificador é considerado o **relacionamento identificador**. O relacionamento identificador é muitos-para-um do conjunto de entidades fraco para o conjunto de entidades identificador, e a participação do conjunto de entidades fraco no relacionamento é total.

Em nosso exemplo, o conjunto de entidades identificador para *pagamento* é *empréstimo*, e um relacionamento *pagamento_empréstimo* que associa entidades *pagamento* a suas entidades *empréstimo* correspondentes é o relacionamento identificador.

Embora um conjunto de entidades fraco não tenha uma chave primária, precisamos de uma maneira de distinguir entre todas essas entidades no conjunto de entidades fraco que depende de uma determinada entidade forte. O **discriminador** de um conjunto de entidades fraco é um conjunto de atributos que permite que essa distinção seja feita. Por exemplo, o discriminador do conjunto de entidades fraco *pagamento* é o atributo *número_pagamento*, já que, para cada empréstimo, um número de pagamento identifica unicamente um pagamento para esse empréstimo. O discriminador de um conjunto de entidades fraco também é chamado a *chave parcial* do conjunto de entidades.

A chave primária de um conjunto de entidades fraco é formada pela chave primária do conjunto de entidades identificador mais o discriminador do conjunto de entidades fraco. No caso do conjunto de entidades *pagamento*, sua chave primária é {*número_empréstimo*, *número_pagamento*}, onde *número_empréstimo* é a chave primária do conjunto de entidades identificador, a saber, *empréstimo*, e *número_pagamento* distingue entidades *pagamento* dentro do mesmo empréstimo.

O conjunto de relacionamento identificador não deve ter atributos descritivos, já que quaisquer atributos podem estar associados ao conjunto de entidades fraco (veja a discussão sobre mudar atributos de conjunto de relacionamento para conjuntos de entidades participantes na seção "Cardinalidades de mapeamento").

Um conjunto de entidades fraco pode participar em relacionamentos que não o relacionamento identificador. Por exemplo, a entidade *pagamento* poderia participar em um relacionamento com o conjunto de entidades *conta*, identificando a conta da qual o pagamento foi feito. Um conjunto de entidades fraco pode participar como proprietário em um relacionamento identificador com outro conjunto de entidades fraco. Também é possível ter um conjunto de entidades fraco com mais de um conjunto de entidades identificador. Uma determinada entidade fraca, então, seria identificada por uma combinação de entidades, uma de cada conjunto de entidades identificador. A chave primária do conjunto de entidades fraco consistiria na união das chaves primárias dos conjuntos de entidades identificadores com o discriminador do conjunto de entidades fraco.

Nos diagramas E-R, um retângulo de contorno duplo indica um conjunto de entidades fraco, e um losango de contorno duplo indica o relacionamento identificador correspondente. Na Figura 6.19, o conjunto de entidades fraco *pagamento* depende do conjunto de entidades forte *empréstimo* por meio do conjunto de relacionamento *pagamento_empréstimo*.

A figura também ilustra o uso de linhas duplas para indicar *participação total* – a participação do conjunto de entidades (fraco) *pagamento* no relacionamento *pagamento_empréstimo* é total, significando que cada pagamento precisa estar relacionado por *pagamento_empréstimo* a algum empréstimo. Finalmente, a seta de *pagamento_empréstimo* até *empréstimo* indica que cada pagamento é para um único empréstimo. O discriminador de um conjunto de entidades fraco é sublinhado, mas com uma linha tracejada, em vez de contínua.

Em alguns casos, o projetista de banco de dados pode escolher expressar um conjunto de entidades fraco como um atributo composto de valores múltiplos do conjunto de entidades proprietário. Em nosso exemplo, essa alternativa exigiria que o conjunto de entidades *empréstimo* tivesse um atributo composto de valores múltiplos *pagamento* consistindo em *número_pagamento*, *data_pagamento* e *quantia_pagamento*. Um conjunto de entidades fraco pode ser modelado de maneira mais apropriada como um atributo se ele participar apenas do relacionamento identificador e se tiver poucos atributos. Por outro lado, uma representação de conjunto de entidades fraco modelará mais apropriadamente uma situação em que o conjunto participe de relacionamentos que não o relacionamento identificador e o conjunto de entidades fraco tenha vários atributos.

Como outro exemplo de um conjunto de entidades que pode ser modelado como um conjunto de entidades fraco,

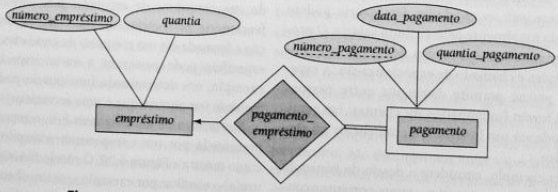


Figura 6.19 Diagrama E-R com um conjunto de entidades fraco.

considere a oferta de um curso em uma universidade. O mesmo curso pode ser oferecido em diferentes semestres, e dentro de um semestre pode haver várias seções para o mesmo curso. Portanto, podemos criar um conjunto de entidades fraco *oferta_curso*, dependente de existência de *curso*; as diversas ofertas do mesmo curso são identificadas por um *semestre* e um *número_seção*, que formam um discriminador mas não uma chave primária.

Recursos de E-R estendidos

Embora os conceitos básicos de E-R possam modelar a maioria dos recursos de banco de dados, alguns aspectos de um banco de dados podem ser expressos mais apropriadamente por certas extensões ao modelo E-R básico. Nesta seção, examinaremos os recursos de E-R estendidos da especialização, generalização, conjuntos de entidades de nível superior e inferior, herança de atributo e agregação.

Especialização

Um conjunto de entidades pode incluir subagrupamentos de entidades que sejam, de algum modo, distintas de outras entidades no conjunto. Por exemplo, um subconjunto de entidades dentro de um conjunto de entidades pode ter atributos que não são compartilhados por todas as entidades do conjunto de entidades. O modelo E-R fornece um meio de representar esses agrupamentos de entidade distintivos.

Como exemplo, considere um conjunto de entidades *pessoa*, com atributos *id_pessoa*, *nome*, *rua* e *cidade*. Uma pessoa pode ser subclassificada como:

- *cliente*
- *funcionário*

Cada um desses tipos de pessoa é descrito por um conjunto de atributos que inclui todos os atributos do conjunto de entidades *pessoa*, além, possivelmente, de atributos adicionais. Por exemplo, as entidades *cliente* podem ser descritas adicionalmente por um atributo *avaliação_crédito*, enquanto as entidades *funcionário* podem ser descritas adicionalmente pelo atributo *salário*. O processo de designar subagrupamentos dentro de um conjunto de entidades é chamado de especialização. A especialização de *pessoa* permite distinguir entre pessoas conforme elas forem funcionários ou clientes; em geral, uma pessoa pode ser um funcionário, um cliente ou ambos, ou nenhum.

Como outro exemplo, considere o desejo do banco de dividir as contas em duas categorias – conta-corrente e conta de poupança. As contas de poupança precisam de um saldo mínimo, mas o banco pode definir taxas de juros diferentemente para clientes distintos, oferecendo melhores taxas para clientes preferenciais. Contas-correntes têm uma taxa de juros fixa, mas oferecem o benefício do saque a descoberto; a quantia do saque a descoberto em uma conta-corrente precisa ser registrada. Cada um desses tipos de conta é descrito por um conjunto de atributos que inclui todos os atributos do conjunto de entidades *conta* mais atributos adicionais.

O banco poderia criar duas especializações de *conta*, a saber, *conta_poupança* e *conta_corrente*. Como vimos anteriormente, as entidades de conta são descritas pelos atributos *número_conta* e *saldo*. O conjunto de entidades *conta_poupança* teria todos os atributos de *conta* e um atributo adicional *taxa_juros*. O conjunto de entidades *conta_corrente* teria todos os atributos de *conta* e um atributo adicional *quantia_saque_descoberto*.

Podemos aplicar especialização repetidamente para refinar um esquema de projeto. Por exemplo, funcionários de banco podem ser subclassificados como um dos seguintes:

- *diretor*
- *caixa*
- *secretária*

Cada um desses tipos de funcionário é descrito por um conjunto de atributos que inclui todos os atributos do conjunto de entidades *funcionário*, além de atributos adicionais. Por exemplo, as entidades *diretor* podem ser descritas adicionalmente pelo atributo *número_diretor*; as entidades *caixa*, pelos atributos *número_caixa* e *horas_semanais*; e as entidades *secretária*, pelo atributo *horas_semanais*. Além disso, as entidades *secretária* podem participar em um relacionamento *secretária_para*, que identifica os funcionários que são auxiliados por uma secretária.

Um conjunto de entidades pode ser especializado por mais de um recurso distintivo. Em nosso exemplo, o recurso distintivo entre as entidades funcionário é o cargo que o funcionário ocupa. Outra especialização coexistente poderia ser baseada em se a pessoa é um funcionário temporário (tempo limitado) ou um funcionário permanente, resultando nos conjuntos de entidades *funcionário_temporário* e *funcionário_permanente*. Quando mais de uma especialização é formada em um conjunto de entidades, uma entidade específica pode pertencer a várias especializações. Por exemplo, um determinado funcionário pode ser um funcionário temporário que é uma secretária.

Em termos de um diagrama E-R, a especialização é representada por um componente *triângulo* rotulado ISA, como mostra a Figura 6.20. O rótulo ISA significa "is a" – é um(a) – e indica, por exemplo, que um cliente "é uma" pessoa. O relacionamento ISA também pode ser chamado de

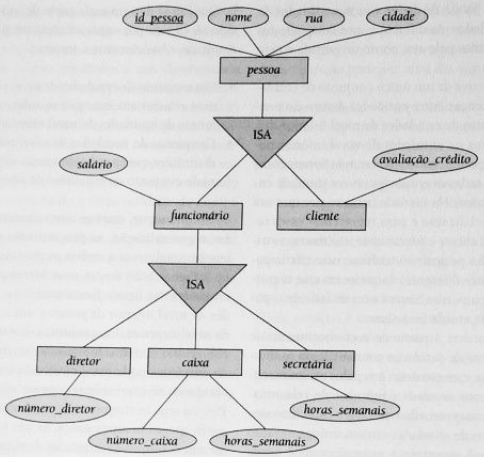


Figura 6.20 Especialização e generalização.

um relacionamento **superclasse-subclasse**. Conjuntos de entidades de nível superior e inferior são descritos como conjuntos de entidades regulares – ou seja, como retângulos contendo o nome do conjunto de entidades.

Generalização

O refinamento de um conjunto de entidades inicial em sucessivos níveis de subagrupamentos de entidade representa um processo de projeto de **cima para baixo (top down)**, em que múltiplos conjuntos de entidade são sintetizados em um conjunto de entidades de nível superior na base dos recursos comuns. O projetista de banco de dados pode ter primeiro identificado um conjunto de entidades *cliente* com os atributos *id_cliente*, *nome_cliente*, *rua_cliente*, *cidade_cliente* e *avaliação_crédito*, e um conjunto de entidades *funcionário* com os atributos *id_funcionário*, *nome_funcionário*, *rua_funcionário*, *cidade_funcionário* e *salário_funcionário*.

Existem semelhanças entre o conjunto de entidades *cliente* e o conjunto de entidades *funcionário* no sentido de que eles possuem vários atributos que são conceitualmente os mesmos nos dois conjuntos de entidades; a saber, os atributos identificadores nome, rua e cidade. Essa semelhança pode ser expressa por **generalização**, que é um relacionamento de contenção que existe entre um conjunto de entidades de *nível superior* e um ou mais conjuntos de entidades de *nível inferior*. Em nosso exemplo, *pessoa* é o conjunto de entidades de nível superior e *cliente* e *funcionário* são conjuntos de entidades de nível inferior. Nesse caso, atributos que são conceitualmente os mesmos tiveram nomes diferentes nos dois conjuntos de entidades de nível inferior. Para criar uma generalização, os atributos precisam receber um nome comum e ser representados com a entidade de nível superior *pessoa*. Podemos usar os nomes de atributo *id_pessoa*, *rua* e *cidade*, como vimos no exemplo da seção "Especialização".

Conjuntos de entidades de nível superior e inferior também podem ser designados pelos termos **superclasse** e **subclasse**, respectivamente. O conjunto de entidades *pessoa* é a superclasse das subclasses *cliente* e *funcionário*.

Para todos os fins práticos, a generalização é uma simples inversão da especialização. Aplicaremos os dois processos, em conjunto, durante o projeto do esquema E-R para uma empresa. Em termos do diagrama E-R propriamente dito, não distinguimos entre especialização e generalização. Novos níveis da representação de entidade serão distinguidos (especialização) ou sintetizados (generalização) quando o esquema de projeto chegar a expressar total-

mente a aplicação de banco de dados e as necessidades de usuário do banco de dados. As diferenças nos dois métodos podem ser caracterizadas pelo seu ponto de partida e pelo objetivo geral.

A especialização deriva de um único conjunto de entidades; ela enfatiza diferenças entre entidades dentro do conjunto criando conjuntos de entidades de nível inferior distintos. Esses conjuntos de entidades de nível inferior podem ter atributos ou podem participar em relacionamentos que não se aplicam a todas as entidades no conjunto de entidades de nível superior. Na verdade, a razão por que um projetista aplica especialização é para representar esses recursos distintivos. Se *cliente* e *funcionário* não tiverem atributos que as entidades *pessoa* não tenham, nem participarem de relacionamentos diferentes daqueles em que as entidades *pessoa* participam, não haverá necessidade de especializar o conjunto de entidades *pessoa*.

A generalização ocorre a partir do reconhecimento de que diversos conjuntos de entidades compartilham os mesmos recursos (ou seja, eles são descritos pelos mesmos atributos e participam nos mesmos conjuntos de relacionamentos). Na base de suas semelhanças, a generalização sintetiza esses conjuntos de entidades em um único conjunto de entidades de nível superior. A generalização é usada para enfatizar as semelhanças entre conjuntos de entidades de nível inferior e para ocultar as diferenças; ela também permite uma economia de representação em que atributos compartilhados não são repetidos.

Herança de atributo

Uma propriedade fundamental das entidades de nível superior e inferior criadas pela especialização e generalização é a **herança de atributo**. Dizemos que os atributos dos conjuntos de entidades de nível superior são **herdados** pelos conjuntos de entidades de nível inferior. Por exemplo, *cliente* e *funcionário* herdam os atributos de *pessoa*. Portanto, *cliente* é descrito pelos seus atributos *nome*, *rua* e *cidade* e, adicionalmente, um atributo *id_cliente*; *funcionário* é descrito pelos seus atributos *nome*, rua e *cidade* e, adicionalmente, atributos *id_funcionário* e *salário*.

Um conjunto de entidades de nível inferior (ou subclasse) também herda participação nos conjuntos de relacionamentos em que sua entidade de nível superior (ou superclasse) participa. Os conjuntos de entidades *diretor*, *caixa* e *secretária* podem participar no conjunto de relacionamentos *trabalha_para*, já que a superclasse *funcionário* participa no relacionamento *trabalha_para*. A herança de atributo se aplica a todas as camadas dos conjuntos de entidades de nível inferior. Os conjuntos de entidades anteriores podem participar em quaisquer relacionamentos em que o conjunto de entidades *pessoa* participa.

Quer uma determinada parte de um modelo E-R tenha sido alcançada por especialização ou por generalização, o resultado é basicamente o mesmo:

- Um conjunto de entidades de nível superior com atributos e relacionamentos que se aplica a todos os seus conjuntos de entidades de nível inferior
- Conjuntos de entidades de nível inferior com recursos distintivos que se aplicam apenas dentro de um determinado conjunto de entidades de nível inferior

No que segue, embora normalmente nos refiramos apenas a generalização, as propriedades que discutimos pertencem totalmente a ambos os processos.

A Figura 6.20 ilustra uma **hierarquia** de conjuntos de entidades. Na figura, *funcionário* é um conjunto de entidades de nível inferior de *pessoa* e um conjunto de entidades de nível superior dos conjuntos de entidades *diretor*, *caixa* e *secretária*. Em uma hierarquia, um determinado conjunto de entidades pode estar envolvido como um conjunto de entidades de nível inferior apenas em um relacionamento ISA; ou seja, os conjuntos de entidades nesse diagrama possuem apenas **herança única**. Se um conjunto de entidades for um conjunto de entidades de nível inferior em mais de um relacionamento ISA, então, o conjunto de entidades possui **herança múltipla** e a estrutura resultante é chamada de uma *treliça*.

Restrições em generalizações

Para modelar uma empresa mais fielmente, o projetista de banco de dados pode escolher impor certas restrições sobre uma determinada generalização. Um tipo de restrição envolve determinar que entidades podem ser membros de um determinado conjunto de entidades de nível inferior. Essa associação pode ser uma das seguintes:

- Definida por condição. Nos conjuntos de entidades de nível inferior definidos por condição, a associação é avaliada na base de se uma entidade satisfaz ou não uma condição ou predicado explícito. Por exemplo, considere que o conjunto de entidades de nível superior *conta* possui o atributo *tipo_conta*. Todas as entidades *conta* são avaliadas no atributo *tipo_conta* definidor. Apenas as entidades que satisfazem a condição *tipo_conta* = "conta de poupança" podem pertencer ao conjunto de entidades de nível inferior *conta_poupança*. Todas as entidades que satisfazem a condição *tipo_conta* = "conta-corrente" são incluídas em *conta_corrente*. Como todas as entidades de nível inferior são avaliadas na base do mesmo atributo (nesse caso, *tipo_conta*), dizemos que esse tipo de generalização é definido por atributo.

- **Definida pelo usuário**. Os conjuntos de entidades de nível inferior definidos pelo usuário não são restritos por uma condição de associação; em vez disso, o usuário de banco de dados atribui entidades a um determinado conjunto de entidades. Por exemplo, vamos considerar que, após três meses de emprego, os funcionários do banco são atribuídos a uma de quatro equipes de trabalho. Assim, representamos as equipes como quatro conjuntos de entidades de nível inferior do conjunto de entidades *funcionário* de nível superior. Um determinado funcionário não é atribuído a uma entidade de equipe específica automaticamente na base de uma condição definidora explícita. Em vez disso, o usuário incumbido dessa decisão faz a atribuição de equipe individualmente. A atribuição é implementada por uma operação que acrescenta uma entidade a um conjunto de entidades.

Um segundo tipo de restrição especifica se ou não as entidades podem pertencer a mais de um conjunto de entidades de nível inferior dentro de uma única generalização. Os conjuntos de entidades de nível inferior podem ser um dos seguintes:

- **Disjuntos**. Uma *restrição de disjunção* exige que uma entidade pertença a não mais que um conjunto de entidades de nível inferior. Em nosso exemplo, uma entidade *conta* pode satisfazer apenas a uma condição para o atributo *tipo_conta*; uma entidade pode ser uma conta de poupança ou uma conta-corrente, mas não ambas.
- **Superpostos**. Nas *generalizações superpostas*, a mesma entidade pode pertencer a mais de um conjunto de entidades de nível inferior dentro de uma única generalização. Como ilustração, considere o exemplo da equipe de trabalho de funcionários, e considere que certos gerentes participam em mais de uma equipe de trabalho. Um determinado funcionário pode, portanto, aparecer em mais de um dos conjuntos de entidades equipe que são conjuntos de entidades de nível inferior de *funcionário*. Portanto, a generalização é superposta.

Como outro exemplo, suponha que a generalização aplicada aos conjuntos de entidades *cliente* e *funcionário* leve a um conjunto de entidades de nível superior *pessoa*. A generalização será superposta se um funcionário também puder ser um cliente.

A superposição de entidade de nível inferior é o caso padrão; uma restrição de disjunção precisa ser colocada explicitamente em uma generalização (ou especialização). Podemos notar uma restrição de disjunção em um diagrama E-R acrescentando a palavra *disjunção* ao lado do símbolo de triângulo.

Uma última restrição, a **restrição de integralidade** em uma generalização ou especialização especifica se uma entidade no conjunto de entidades de nível superior precisa ou não pertencer a pelo menos um dos conjuntos de entidades de nível inferior com a generalização/especialização. Essa restrição pode ser uma das seguintes:

- **Generalização ou especialização total**. Cada entidade de nível superior precisa pertencer a um conjunto de entidades de nível inferior.
- **Generalização ou especialização parcial**. Algumas entidades de nível superior podem não pertencer a um conjunto de entidades de nível inferior.

A generalização parcial é o padrão. Podemos especificar generalização total em um diagrama E-R usando uma linha dupla para conectar o retângulo representando o conjunto de entidades de nível superior com o símbolo de triângulo. (Essa notação é semelhante a notação para a participação total em um relacionamento).

A generalização *conta* é total: todas as entidades de conta precisam ser uma conta de poupança ou uma conta-corrente. Como o conjunto de entidades de nível superior obtido por generalização normalmente é composto apenas pelas entidades nos conjuntos de entidades de nível inferior, a restrição de integralidade para um conjunto de entidades de nível superior generalizado normalmente é total. Quando a generalização é parcial, uma entidade de nível superior não está restrita a aparecer em um conjunto de entidades de nível inferior. Os conjuntos de entidades de equipe de trabalho ilustram uma especialização parcial. Como os funcionários são atribuídos a uma equipe somente após três meses no cargo, algumas entidades *funcionário* podem não ser membros de quaisquer dos conjuntos de entidades de equipe de nível inferior.

Podemos caracterizar mais completamente os conjuntos de entidades de equipe como uma especialização superposta parcial de *funcionário*. A generalização de *conta_corrente* e *conta_poupança* em *conta* é uma generalização disjunta total. As restrições de integralidade e disjunção, no entanto, não dependem uma da outra. Os padrões de restrição também podem ser disjuntos parciais ou superpostos totais.

Podemos ver que certas necessidades de inserção e exclusão seguem as restrições que se aplicam a uma determinada generalização ou especialização. Por exemplo, quando uma restrição de integralidade total está em vigor, uma entidade inserida em um conjunto de entidades de nível superior também precisa estar inserida em pelo menos um dos conjuntos de entidades de nível inferior. Com uma restrição definida por condição, todas as entidades de nível superior que satisfazem a condição precisam estar inseridas nesse conjunto de entidades de nível inferior. Finalmente, uma entidade que é excluída de um conjunto de entidades de nível superior também é excluída de todos os conjuntos de entidades de nível inferior associados aos quais ela pertence.

Agregação

Uma limitação do modelo E-R é que ele não pode expressar relacionamentos entre relacionamentos. Para ilustrar a necessidade dessa construção, considere o relacionamento ternário *trabalha_em*, que vimos anteriormente, entre um *funcionário*, uma *agência* e um *cargo* (veja a Figura 6.12). Agora, suponha que queremos registrar gerentes para tarefas realizadas por um funcionário em uma agência; ou seja, queremos registrar gerentes para combinações (*funcionário, agência, cargo*). Vamos considerar que existe um conjunto de entidades *gerente*.

Uma alternativa para representar esse relacionamento é criar um relacionamento quaternário *gerencia* entre *funcionário, agência, cargo* e *gerente*. (Um relacionamento quaternário é necessário – um relacionamento binário entre *gerente* e *funcionário* não nos permitiria representar quais combinações (*agência, cargo*) de um funcionário são gerenciadas por qual gerente.) Usando as restrições de modelagem E-R básicas, obtemos o diagrama E-R da Figura 6.21. (Omitimos os atributos dos conjuntos de entidades para simplificação.)

Parece que os conjuntos de relacionamentos *trabalha_em* e *gerencia* podem ser combinados em um único conjunto de relacionamentos. Entretanto, não devemos combiná-los em um único relacionamento, já que algumas combinações *funcionário, agência, cargo* podem não ter um gerente.

Contudo, existem informações redundantes na figura resultante, uma vez que toda combinação *funcionário, agência, cargo* em *gerencia* também está em *trabalha_em*. Se o gerente fosse um valor em vez de uma entidade *gerente*, poderíamos tornar *gerente* um atributo de valores múltiplos do relacionamento *trabalha_em*. Isso, porém, dificulta (logicamente em custo de execução) encontrar, por exemplo, triplas funcionário-agência-cargo pelas quais um gerente é responsável. Como o gerente é uma entidade *gerente*, essa alternativa é descartada em qualquer caso.

A melhor maneira de modelar uma situação como a que acabamos de descrever é usando agregação. A agregação é uma abstração pela qual os relacionamentos são tratados como entidades de nível superior. Portanto, para nosso exemplo, consideramos o conjunto de relacionamentos *trabalha_em* (relacionando os conjuntos de entidades *funcionário, agência* e *cargo*) como um conjunto de entidades de nível superior chamado *trabalha_em*. Esse conjunto de entidades é tratado da mesma maneira que qualquer outro conjunto de entidades. Podemos, então, criar um relacionamento binário *gerencia* entre *trabalha_em* e *gerente* para representar quem gerencia quais tarefas. A Figura 6.22 mostra uma notação para agregação comumente usada para representar essa situação.

Notações de E-R alternativas

A Figura 6.22 resume os símbolos que usamos nos diagramas E-R. Não há um padrão universal para notação de diagrama E-R, e os diversos livros e softwares de diagrama E-R usam diferentes notações.

A Figura 6.24 indica algumas notações alternativas que são amplamente usadas. Um conjunto de entidades pode ser representado como um retângulo com o nome do lado de fora, e os atributos listados um abaixo do outro dentro do retângulo. Os atributos de chave primária são indicados listando-os no alto, com uma linha separando-os dos outros atributos.

As restrições de cardinalidade podem ser indicadas de várias maneiras diferentes, como mostra a Figura 6.24. Algu-

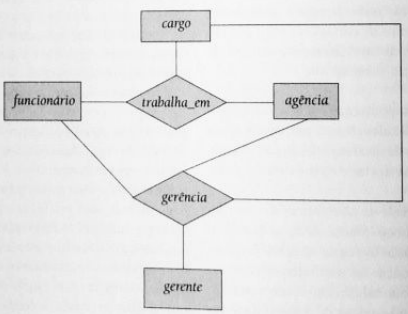


Figura 6.21 Diagrama E-R com relacionamentos redundantes.

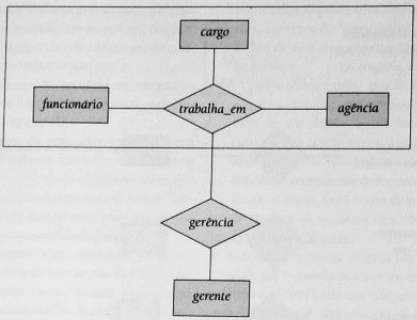


Figura 6.22 Diagrama E-R com agregação.

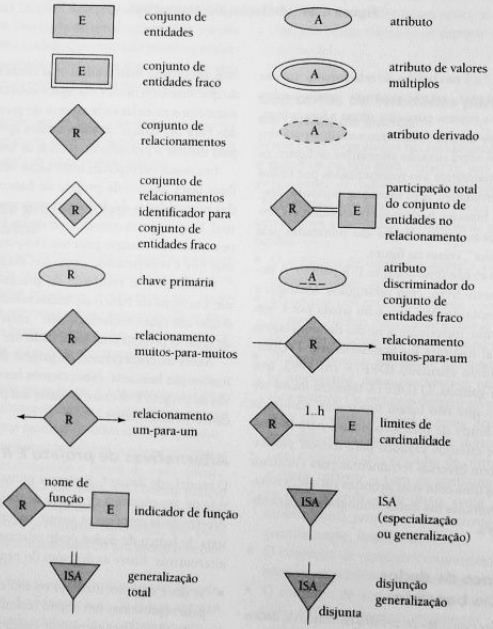


Figura 6.23 Símbolos usados na notação E-R.

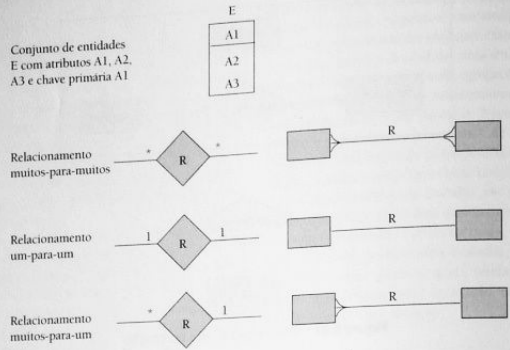


Figura 6.24 Notações E-R alternativas.

mas vezes, os rótulos * e 1 nas bordas do relacionamento são usados para representar relacionamentos muitos-para-muitos, um-para-um e muitos-para-um, como a figura mostra. O caso de um-para-muitos é simétrico a muitos-para-um e não é mostrado. Em outra notação alternativa na figura, os conjuntos de relacionamentos são representados por linhas entre conjuntos de entidades, sem losangos; portanto, apenas relacionamentos binários podem ser modelados. As restrições de cardinalidade nessa notação são mostradas pela notação "pé-de-galinha", como na figura.

Infelizmente, não existe uma notação E-R padrão. A notação que usamos neste livro, com retângulos, losangos e elipses, é chamada notação de Chen e foi usada por Chen em seu documento que introduziu a noção da modelagem E-R. O U.S. National Institute for Standards and Technology definiu um padrão chamado IDEF1X em 1993, que usa a notação pé-de-galinha. O IDEF1X também inclui várias outras notações que não foram mostradas, incluindo barras verticais na borda de relacionamento, para indicar participação total, e círculos vazados para indicar participação parcial. Existem diversas ferramentas para construir diagramas E-R, cada uma com suas próprias variantes notacionais. Veja as referências nas notas bibliográficas para obter mais informações.

Projeto de banco de dados para instituição bancária

Agora veremos as necessidades do projeto de banco de dados de uma instituição bancária em mais detalhes, e desenvolveremos um projeto mais realista, mas também mais complicado, do que vimos em nossos exemplos anteriores. Entretanto, não tentaremos modelar cada aspecto do projeto de banco de dados para um banco; consideraremos apenas alguns aspectos para ilustrar o processo do projeto de banco de dados.

Em nosso exemplo de instituição bancária, aplicamos as duas fases iniciais de projeto de banco de dados: a coleta das necessidades de dados e o projeto do esquema conceitual. Empregamos o modelo de dados E-R para traduzir necessidades de usuário para um esquema de projeto conceitual que é representado como um diagrama E-R.

Finalmente, o resultado do processo de projeto E-R é um esquema de banco de dados relacional. Na seção "Redução aos esquemas relacionais", consideramos o processo de gerar o projeto relacional dado um projeto E-R.

Antes de começarmos no projeto de banco de dados de instituição bancária, esboçaremos brevemente as alternativas de projeto E-R entre as quais um projetista de banco de dados pode escolher.

Alternativas de projeto E-R

O modelo de dados E-R oferece muita flexibilidade no projeto de um esquema de banco de dados para modelar uma determinada empresa. A seguir, sugerimos como um projetista de banco de dados pode selecionar da ampla faixa de alternativas. Entre as decisões do projetista estão:

- Se deve usar um atributo ou um conjunto de entidades para representar um objeto (discutido anteriormente na seção "Uso de conjuntos de entidades versus atributos")

- Se um conceito do mundo real é expresso mais precisamente por um conjunto de entidades ou por um conjunto de relacionamentos (seção "Uso de conjuntos de entidades *versus* conjuntos de relacionamento")
- Se deve usar um relacionamento ternário ou um par de relacionamentos binários (seção "Conjuntos de relacionamento binários *versus* enários")
- Se deve usar um conjunto de entidades forte ou fraco (seção "Conjuntos de entidades fracos"); um conjunto de entidades forte e seu conjunto de entidades fraco dependente podem ser considerados como um único "objeto" no banco de dados, já que as entidades fracas dependem da existência de uma entidade forte
- Se usar generalização (seção "Generalização") é apropriado; a generalização, ou uma hierarquia de relacionamentos ISA, contribui para modularidade permitindo que atributos comuns de conjuntos de entidades semelhantes sejam representados em um local em um diagrama E-R
- Se usar agregação (seção "Agregação") é apropriado; a agregação agrupa uma parte de um diagrama E-R em um único conjunto de entidades, permitindo tratar o conjunto de entidades agregado como uma única unidade sem preocupação com os detalhes de sua estrutura interna.

Veremos que o projetista de banco de dados precisa ter um bom conhecimento da empresa sendo modelada para tomar as várias decisões de projeto necessárias.

Necessidades de projeto para o banco de dados de banco

A especificação inicial das necessidades de usuário pode ser baseada em entrevistas com os usuários do banco de dados e na análise do próprio projetista. A descrição que surge dessa fase do projeto serve como base para especificar a estrutura conceitual do banco de dados. Aqui estão as principais características da instituição de banco.

- O banco é organizado em agências. Cada agência está localizada em uma determinada cidade e é identificada por um nome único. O banco monitora o ativo de cada agência.
- Os clientes do banco são identificados por seus valores *id_cliente*. O banco armazena o nome de cada cliente, além da rua e cidade onde ele mora. Os clientes podem ter contas e podem fazer empréstimos. Um cliente pode estar associado a um determinado banqueiro, que pode agir como gerente de empréstimo ou um banqueiro pessoal para esse cliente.
- Os funcionários do banco são identificados por seus valores *id_funcionário*. A administração do banco armazena o nome e número de telefone de cada funcionário, os nomes dos dependentes do funcionário e o *id_funcionário* do gerente do funcionário. O banco também controla a data de admissão do funcionário e, portanto, o tempo de serviço.
- O banco oferece dois tipos de conta – conta de poupança e conta-corrente. As contas podem ser mantidas por mais de um cliente, e um cliente pode ter mais de uma conta. Cada conta recebe um número de conta único. O banco mantém um registro do saldo de cada conta e a data mais recente em que a conta foi acessada pelo cliente. Além disso, cada conta de poupança possui uma taxa de juros, e os saques a descoberto são registrados para cada conta-corrente.
- Um empréstimo se origina em uma agência específica e pode ser mantido por um ou mais clientes. Um empréstimo é identificado por um número de empréstimo único. Para cada empréstimo, o banco controla os pagamentos às contas de empréstimo. Como as necessidades de modelagem para esse controle são semelhantes e gostaríamos de manter nossa aplicação de exemplo pequena, não controlamos esses depósitos e retiradas em nosso modelo.

Conjuntos de entidades para o banco de dados de banco

Nossa especificação das necessidades de dados serve como o ponto de partida para construir um esquema conceitual para o banco de dados. Das características listadas na seção "Necessidades de projeto para o banco de dados de banco", começamos a identificar conjuntos de entidades e seus atributos:

- O conjunto de entidades *agência*, com atributos *nome_agência*, *cidade_agência* e *ativo*.
- O conjunto de entidades *cliente*, com atributos *id_cliente*, *nome_cliente*, *rua_cliente* e *cidade_cliente*. Um atributo adicional possível é *nome_banqueiro*.
- O conjunto de entidades *funcionário*, com atributos *id_funcionário*, *nome_funcionário*, *número_telefone*, *salário* e *gerente*. Os recursos descritivos adicionais são o atributo de valores múltiplos *nome_dependente*, o atributo base *data_admissão* e o atributo derivado *tempo_serviço*.
- Dois conjuntos de entidades de conta – *conta_poupança* e *conta_corrente* – com os atributos comuns de *número_conta* e *saldo*; além disso, *conta_poupança* possui o atributo *taxa_juros* e *conta_corrente* possui o atributo *quantia_saque_descoberto*.
- O conjunto de entidades *empréstimo*, com os atributos *número_empréstimo*, *quantia* e *agência_origem*.
- O conjunto de entidades fraco *pagamento_empréstimo*, com atributos *número_pagamento*, *data_pagamento* e *quantia_pagamento*.

Conjuntos de relacionamentos para o banco de dados de banco

Agora voltaremos ao esquema de projeto rudimentar da seção "Conjuntos de entidades para o banco de dados de banco" e especificamos os seguintes conjuntos de relacionamentos e cardinalidades de mapeamento. No processo, também refinamos algumas das decisões que tomamos anteriormente com relação aos atributos dos conjuntos de entidades.

- *tomador*, um conjunto de relacionamentos muitos-para-muitos entre *cliente* e empréstimo.
- *agência_empréstimo*, um conjunto de relacionamentos muitos-para-um que indica em que agência um empréstimo foi originado. Note que esse conjunto de relacionamentos substitui o atributo *agência_origem* do conjunto de entidades *empréstimo*.
- *pagamento_empréstimo*, um relacionamento um-para-muitos de *empréstimo* para *pagamento*, que documenta que um pagamento é feito em um empréstimo.
- *depositante*, com atributo de relacionamento *data_acesso*, um conjunto de relacionamentos muitos-para-muitos entre *cliente* e *conta*, indicando que um cliente possui uma conta.
- *banqueiro_cliente*, com atributo de relacionamento *tipo*, um conjunto de relacionamentos muitos-para-um indicando que um cliente pode ser aconselhado por um funcionário do banco, e que um funcionário do banco pode aconselhar um ou mais clientes. Observe que esse conjunto de relacionamentos substituiu o atributo *nome_banqueiro* do conjunto de entidades *cliente*.
- *trabalha_para*, um conjunto de relacionamentos entre entidades *funcionário* com indicadores de função (papel) *gerente* e *trabalhador*; as cardinalidades de mapeamento indicam que um funcionário trabalha para um único gerente e que um gerente supervisiona um ou mais funcionários. Note que esse conjunto de relacionamentos substituiu o atributo *gerente* de *funcionário*.

Diagrama E-R para o banco de dados de banco

Utilizando as discussões na seção "Conjuntos de relacionamentos para o banco de dados de banco", agora apresentamos o diagrama E-R completo para nossa instituição bancária de exemplo. A Figura 6.25 ilustra a representação completa de um modelo conceitual de um banco, ex-

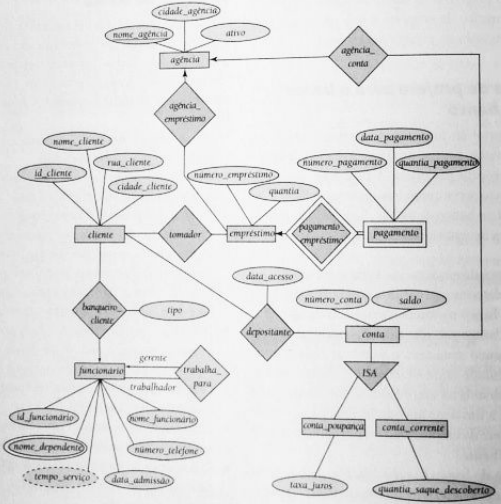


Figura 6.25 Diagrama E-R para uma instituição bancária.

número_empréstimo	*Quantia*
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Figura 6.26 Tabela *empréstimo*.

presso em termos de conceitos E-R. O diagrama inclui os conjuntos de entidades, atributos, conjuntos de relacionamentos e cardinalidades de mapeamento obtidos pelos processos de projeto das seções "Necessidades de projeto para o banco de dados de banco" e "Conjuntos de entidades para o banco de dados de banco", e refinados na seção "Conjuntos de relacionamentos para o banco de dados de banco".

O diagrama E-R para nossa visão simplificada de uma instituição bancária já está bastante complexo. Os diagramas E-R para empresas reais não podem ser desenhados em uma única página e precisam ser desmembrados em várias partes. As entidades podem precisar aparecer várias vezes, em diferentes partes do diagrama. Os atributos da entidade são mostrados em uma ocorrência da entidade (preferivelmente a primeira ocorrência), e todas as outras ocorrências da entidade são mostradas sem quaisquer atributos.

Redução aos esquemas relacionais

Podemos representar um banco de dados que se conforma a um esquema de banco de dados E-R por uma coleção de esquemas de relação. Para cada conjunto de entidades e para cada conjunto de relacionamentos no banco de dados existe um esquema de relação ao qual atribuímos o nome do conjunto de entidades ou conjunto de relacionamentos correspondente.

Tanto o modelo E-R quanto o modelo de banco de dados relacional são representações lógicas e abstratas de empresas do mundo real. Como os dois modelos empregam princípios de projeto semelhantes, podemos converter um projeto E-R em um projeto relacional.

Nesta seção, descrevemos como um esquema E-R pode ser representado por esquemas de relação, e como as restrições surgindo do projeto E-R podem ser mapeadas para restrições em um esquema de relação.

Representação dos conjuntos de entidades fortes

Seja E um conjunto de entidades forte com atributos descritivos $a_1, a_2, \ldots, a_n$. Representamos essa entidade por um esquema chamado E com n atributos distintos. Cada tupla em uma relação nesse esquema corresponde a uma entidade do conjunto de entidades E. (Descrevemos como manipular atributos compostos e de valores múltiplos mais adiante, na seção "Atributos compostos e de valores múltiplos".)

Para esquemas derivados de conjuntos de entidades fortes, a chave primária do conjunto de entidades serve como a chave primária do esquema resultante. Isso porque cada tupla corresponde a uma entidade específica no conjunto de entidades.

Como ilustração, considere o conjunto de entidades *empréstimo* do diagrama E-R na Figura 6.7. Esse conjunto de entidades possui dois atributos: *número_empréstimo* e *quantia*. Representamos esse conjunto de entidades por um esquema chamado *empréstimo*, com dois atributos:

$$empréstimo = (\underline{número\_empréstimo}, quantia)$$

Note que, como *número_empréstimo* é a chave primária do conjunto de entidades, ela também é a chave primária do esquema de relação.

Uma relação nesse esquema é mostrada na Figura 6.26. A tupla

$$(\text{L-17}, 1000)$$

significa que o número de empréstimo L-17 possui uma quantia de empréstimo de $1.000. Podemos acrescentar uma nova entidade ao banco de dados inserindo uma tupla na relação correspondente. Também podemos excluir ou modificar entidades modificando a tupla correspondente.

Representação dos conjuntos de entidades fracos

Seja A um conjunto de entidades fraco com atributos a_1, a_2,..., a_m. Seja B o conjunto de entidades forte do qual A depende. Seja a chave primária de B constituída dos atributos $b_1, b_2,..., b_n$. Representamos o conjunto de entidades A por um esquema de relação chamado A com um atributo para cada membro do conjunto:

$$\{a_1, a_2,..., a_m\} \cup \{b_1, b_2,..., b_n\}$$

Para esquemas derivados de um conjunto de entidades fraco, a combinação da chave primária do conjunto de entidades forte com o discriminador do conjunto de entidades fraco serve como a chave primária do esquema. Além de criar uma chave primária, também criamos uma restrição de chave estrangeira na relação A, especificando que os atributos $b_1, b_2,..., b_n$ referenciam a chave primária da relação B. A restrição de chave estrangeira garante que, para cada tupla representando uma entidade fraca, existe uma tupla equivalente representando a entidade forte correspondente.

Como ilustração, considere o conjunto de entidades *pagamento* no diagrama E-R da Figura 6.19. Esse conjunto de entidades possui três atributos: *número_pagamento*, *data_pagamento* e *quantia_pagamento*. A chave primária do conjunto de entidades *empréstimo*, da qual *pagamento* depende, é *número_empréstimo*. Portanto, representamos *pagamento* por um esquema com quatro atributos:

pagamento = (*número_empréstimo*, *número_pagamento*, *data_pagamento*, *quantia_pagamento*)

A chave primária consiste na chave primária de *empréstimo*, juntamente com o discriminador de *pagamento*, que é *número_pagamento*. Também criamos uma restrição de chave estrangeira no esquema *pagamento*, com o atributo *número_empréstimo* referenciando a chave primária do esquema *empréstimo*.

Representação dos conjuntos de relacionamentos

Seja R um conjunto de relacionamentos; seja $a_1, a_2,..., a_m$ o conjunto dos atributos formados pela união das chaves primárias de cada um dos conjuntos de entidades participando em R; e sejam $b_1, b_2,..., b_n$ os atributos descritivos de R (se houver). Representamos esse conjunto de relacionamentos por um esquema de relação chamado R com um atributo para cada membro do conjunto:

$$\{a_1, a_2,..., a_m\} \cup \{b_1, b_2,..., b_n\}$$

Anteriormente, na seção "Conjuntos de relacionamento", descrevemos como escolher uma chave primária para um conjunto de relacionamentos binário. Como vimos naquela seção, tomar todos os atributos de chave primária de todos os conjuntos de entidades relacionados serve para identificar uma tupla específica, mas, para conjuntos de relacionamentos um-para-um, muitos-para-um e um-para-muitos, isso acaba sendo um conjunto de atributos maior do que precisamos na chave primária. Em vez disso, a chave primária é escolhida desta maneira:

- Para um relacionamento muitos-para-muitos, a união dos atributos de chave primária dos conjuntos de entidades participantes se torna a chave primária.
- Para um conjunto de relacionamentos um-para-um binário, a chave primária de qualquer conjunto de entidades pode ser escolhida como a chave primária para o relacionamento. A escolha do conjunto de entidades dentre os relacionados pelo conjunto de relacionamentos pode ser feita arbitrariamente.
- Para um conjunto de relacionamentos muitos-para-um ou um-para-muitos binário, a chave primária do conjunto de entidades no lado "muitos" do conjunto de relacionamentos serve como a chave primária.
- Para um conjunto de relacionamentos enário sem quaisquer setas em suas bordas, a união dos atributos de chave primária dos conjuntos de entidades participantes se torna a chave primária.
- Para um conjunto de relacionamentos enário com uma seta em uma de suas bordas, as chaves primárias dos conjuntos de entidades que não estejam no lado da "seta" do conjunto de relacionamentos servem como a chave primária para o esquema. Lembre-se de que permitimos apenas uma seta de um conjunto de relacionamento.

Também criamos restrições de chave estrangeira na relação R da maneira a seguir. Para cada conjunto de entidades E_i relacionado ao conjunto de relacionamentos R, criamos uma restrição de chave estrangeira da relacionamento R, com os atributos de R que sejam atributos de chave primária de E referenciando a chave primária da relação representando E_i.

Como ilustração, considere o conjunto de relacionamentos *tomador* no diagrama E-R da Figura 6.7. Esse conjunto de relacionamentos envolve os dois conjuntos de entidades a seguir:

- *cliente*, com a chave primária *id_cliente*
- *empréstimo*, com a chave primária *número_empréstimo*

Como o conjunto de relacionamentos não possui atributos, o esquema *tomador* possui dois atributos:

tomador = (*id_cliente*, *número_empréstimo*)

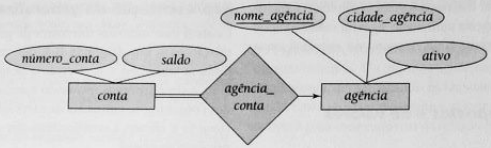


Figura 6.27 Diagrama E-R.

A chave primária para a relação *tomador* é a união dos atributos de chave primária de *cliente* e *empréstimo*. Também criamos duas restrições de chave estrangeira na relação *tomador*, com o atributo *id_cliente* referenciando a chave primária de *cliente* e o atributo *número_empréstimo* referenciando a chave primária de *empréstimo*.

Redundância de esquemas

Um conjunto de relacionamentos vinculando um conjunto de entidades fraco ao conjunto de entidades forte correspondente é tratado de forma especial. Como observamos na seção "Conjuntos de entidades fracos", esses relacionamentos são muitos-para-um e não possuem atributos descritivos. Além disso, a chave primária de um conjunto de entidades fraco inclui a chave primária do conjunto de entidades forte. No diagrama E-R da Figura 6.19, o conjunto de entidades fraco *pagamento* é dependente do conjunto de entidades forte *empréstimo* por meio do conjunto de relacionamentos *pagamento_empréstimo*. A chave primária de *pagamento* é {*número_empréstimo*, *número_pagamento*} e a chave primária de *empréstimo* é {*número_empréstimo*}. Como *pagamento_empréstimo* não tem quaisquer atributos descritivos, o esquema *pagamento_empréstimo* possui dois atributos, *número_empréstimo* e *número_pagamento*. O esquema para o conjunto de entidades *pagamento* possui quatro atributos, *número_empréstimo*, *número_pagamento*, *data_pagamento* e *quantia_pagamento*. Cada combinação (*número_empréstimo*, *número_pagamento*) em uma relação *pagamento_empréstimo* também estaria presente na relação no esquema *pagamento* e vice-versa. Portanto, o esquema *pagamento_empréstimo* é redundante. Em geral, o esquema para o conjunto de relacionamentos vinculando um conjunto de entidades fraco ao seu conjunto de entidades forte correspondente é redundante e não precisa estar presente em um projeto de banco de dados relacional baseado em um diagrama E-R.

Combinação de esquemas

Considere um conjunto de relacionamentos muitos-para-um *AB* do conjunto de entidades *A* para o conjunto de entidades *B*. Usando nosso esquema de construção de esquema relacional descrito anteriormente, obtemos três esquemas: *A*, *B* e *AB*. Suponha ainda que a participação de *A* no relacionamento seja total; isto é, toda entidade *a* no conjunto de entidades *A* precisa participar no relacionamento *AB*. Depois, podemos combinar os esquemas *A* e *AB* para formar um único esquema consistindo na união das colunas de ambos os esquemas.

Como ilustração, considere o diagrama E-R da Figura 6.27. A linha dupla no diagrama E-R indica que a participação de *conta* em *agência_conta* é total. Conseqüentemente, uma conta não pode existir sem estar associada a uma agência específica. Além disso, o conjunto de relacionamentos *agência_conta* é muitos-para-um de *conta* para *agência*. Portanto, podemos combinar o esquema para *agência_conta* com o esquema para *conta* e exigir apenas estes dois esquemas:

- *conta* = (*número_conta*, *saldo*, *nome_agência*)
- *agência* = (*nome_agência*, *cidade_agência*, *ativo*)

No caso dos relacionamentos um-para-um, o esquema de relação para o conjunto de relacionamentos pode ser combinado com os esquemas para qualquer um dos conjuntos de entidades.

Podemos combinar esquemas mesmo se a participação for parcial, usando valores nulos; no exemplo anterior, armazenaríamos valores nulos para o atributo *nome_agência* para contas que não possuem agência associada.

A chave primária do esquema combinado é a mesma do esquema do conjunto de entidades com o qual o esquema do conjunto de relacionamentos foi mesclado. No exemplo anterior, a chave primária é *número_conta*.

O esquema representando o conjunto de relacionamentos teria tido restrições de chave estrangeira referenciando cada um dos conjuntos de entidades participantes no conjunto de relacionamento. Descartamos a restrição referenciando o esquema do conjunto de entidades com o qual o esquema do conjunto de relacionamentos é mesclado, e acrescentamos as outras restrições de chave estrangeira ao

esquema combinado. Em nosso exemplo anterior, a restrição de chave estrangeira com *nome_agência* referenciando *agência* é mantida como uma restrição no esquema *conta* combinado.

Atributos compostos e de valores múltiplos

Manipulamos atributos compostos criando um atributo separado para cada um dos atributos componentes; não criamos um atributo separado para o atributo composto em si. Suponha que *endereço* seja um atributo composto do conjunto de entidades *cliente*, e os componentes de *endereço* sejam *rua* e *cidade*. O esquema gerado de *cliente* contém atributos *rua_endereço* e *cidade_endereço*; não existe um atributo ou esquema separado para *endereço*. Voltaremos a esse assunto na seção "Domínios atômicos e primeira forma normal" do Capítulo 7.

Vimos que os atributos em um diagrama E-R em geral mapeiam diretamente em colunas os esquemas de relação apropriados. Os atributos de valores múltiplos, no entanto, são uma exceção; novos esquemas de relação são criados para esses atributos.

Para um atributo de valores múltiplos M, criamos um esquema de relação R com um atributo A que corresponde a M e atributos correspondentes à chave primária do conjunto de entidades ou conjunto de relacionamentos do qual M é um atributo.

Como ilustração, considere o diagrama E-R na Figura 6.25. O diagrama inclui o conjunto de entidades *funcionário* com um atributo de valores múltiplos *nome_dependente*. A chave primária de *funcionário* é *id_funcionário*. Para esse atributo de valores múltiplos, criamos um esquema de relação

nome_dependente (*id_funcionário*, *nome_dependente*)

Cada dependente de um funcionário é representado como uma tupla única na relação sobre esse esquema. Assim, se tivéssemos um funcionário com *id_funcionário* 12-234 e dependentes John e Mary, a relação *nome_dependente* teria duas tuplas (12-234, John) e (12-234, Mary).

Criamos uma chave primária do esquema de relação consistindo em todos os atributos do esquema. No exemplo anterior, a chave primária consiste em ambos os atributos da relação *nome_dependente*.

Além disso, criamos uma chave estrangeira no esquema de relação, com o atributo gerado da chave primária do conjunto de entidades referenciando a relação gerada do conjunto de entidades. No exemplo anterior, a restrição seria que o atributo *id_funcionário* referencia a relação *funcionário*.

Representação da generalização

Existem dois métodos diferentes de projetar esquemas de relação para um diagrama E-R que inclui generalização. Embora tenhamos nos referido à generalização na Figura 6.20, a simplificamos incluindo apenas a primeira camada dos conjuntos de entidades de nível inferior – ou seja, *funcionário* e *cliente*. Consideramos que *id_pessoa* é a chave primária de *pessoa*.

1. Crie um esquema para o conjunto de entidades de nível superior. Para cada conjunto de entidades de nível inferior, crie um esquema que inclua um atributo para cada um dos atributos desse conjunto de entidades mais um para cada atributo da chave primária do conjunto de entidades de nível superior. Portanto, para o diagrama E-R da Figura 6.20, temos três esquemas:

   pessoa = (*id_pessoa*, *nome*, *rua*, *cidade*)
   funcionário = (*id_pessoa*, *salário*)
   cliente = (*id_pessoa*, *avaliação_crédito*)

   Os atributos de chave primária do conjunto de entidades de nível superior se tornam atributos de chave primária do conjunto de entidades de nível superior, bem como de todos os conjuntos de entidades de nível inferior. Esses podem ser vistos sublinhados no exemplo anterior.

   Além disso, criamos restrições de chave estrangeira nos conjuntos de entidades de nível inferior, com seus atributos de chave primária referenciando a chave primária da relação criada a partir do conjunto de entidades de nível superior. No exemplo anterior, o atributo *id_pessoa* de *funcionário* referenciaria a chave primária de *pessoa*, e do mesmo modo para *cliente*.
2. Uma representação alternativa é possível, se a generalização for disjunta e completa – ou seja, se nenhuma entidade for um membro dos dois conjuntos de entidades de nível inferior diretamente abaixo do conjunto de entidades de nível superior, e se toda entidade no conjunto de entidades de nível superior também for um membro de um dos conjuntos de entidades de nível inferior. Aqui, não criamos um esquema para o conjunto de entidades de nível superior. Em vez disso, para cada conjunto de entidades de nível inferior, crie um esquema que inclua um atributo para cada um dos atributos desse conjunto de entidades mais um para *cada* atributo do conjunto de entidades de nível superior. Depois, para o diagrama E-R da Figura 6.20, temos os dois esquemas:

funcionário = (*id_pessoa, nome, rua, cidade, salário*)
cliente = (*id_pessoa, nome, rua, cidade, avaliação_crédito*)

Esses dois esquemas possuem *id_pessoa*, que é o atributo de chave primária do conjunto de entidades de nível superior *pessoa*, como sua chave primária.

Uma desvantagem do segundo método está em definir restrições de chave primária. Para ilustrar o problema, suponha que temos um conjunto de relacionamentos *R* envolvendo o conjunto de entidades *pessoa*. Com o primeiro método, quando criamos um esquema de relação *R* do conjunto de relacionamento, também definimos uma restrição de chave estrangeira em *R*, referenciando o esquema *pessoa*. Infelizmente, com o segundo método, não temos uma única relação à qual uma restrição de chave estrangeira em *R* possa se referir. Para evitar esse problema, precisamos criar um esquema de relação *pessoa* contendo pelo menos os atributos de chave primária da entidade *pessoa*.

Se o segundo método fosse usado para uma generalização superposta, alguns valores seriam armazenados várias vezes, desnecessariamente. Por exemplo, se uma pessoa é, ao mesmo tempo, um funcionário e um cliente, os valores para *rua* e *cidade* são armazenados duas vezes. Se a generalização não fosse completa – isto é, se alguma pessoa não é nem um funcionário nem um cliente – então, uma tabela extra *pessoa* seria necessária para representar essas pessoas.

Representação de agregação

Projetar esquemas para um diagrama E-R contendo agregação é uma tarefa simples. Considere o diagrama da Figura 6.22. O esquema para o conjunto de relacionamentos *gerencia* entre a agregação de *trabalha_em* e o conjunto de entidades *gerente* inclui um atributo para cada atributo nas chaves primárias do conjunto de entidades *gerente* e o conjunto de relacionamentos *trabalha_em*. Ele também inclui um atributo para quaisquer atributos descritivos, se existirem, do conjunto de relacionamentos *gerencia*. Depois, transformamos os conjuntos de relacionamentos e os conjuntos de entidades dentro da entidade agregada seguindo as regras que já definimos.

As regras que vimos anteriormente para criar restrições de chave primária e de chave estrangeira nos conjuntos de relacionamentos também podem ser aplicadas aos conjuntos de relacionamentos envolvendo agregações, com a agregação tratada como qualquer outra entidade. A chave primária da agregação é a chave primária do seu conjunto de relacionamentos definidor. Nenhuma relação separada é necessária para representar a agregação; em seu lugar, é usada a relação criada do relacionamento definidor.

Esquemas relacionais para instituição bancária

Na Figura 6.25, mostramos um diagrama E-R para uma instituição bancária. O conjunto de esquemas de relação correspondente, gerado usando as técnicas descritas anteriormente nesta seção, é mostrado a seguir. Indicamos a chave primária para cada esquema de relação por um sublinhado.

- Esquemas derivados de uma entidade forte:

agência = (*nome_agência, cidade_agência, ativo*)
cliente = (*id_cliente, nome_cliente, rua_cliente, cidade_cliente*)
empréstimo = (*número_empréstimo, quantia*)
conta = (*número_conta, saldo*)
funcionário = (*id_funcionário, nome_funcionário, número_telefone, data_admissão*)

- Esquemas derivados de um atributo de valor múltiplo: (Não representamos atributos derivados. Eles são definidos em uma view ou função especialmente definida.)

nome_dependente = (*id_funcionário, nome_dependente*)

- Esquemas derivados de um conjunto de relacionamentos envolvendo conjuntos de entidades fortes:

agência_conta = (*número_conta, nome_agência*)
agência_empréstimo = (*número_empréstimo, nome_agência*)
tomador = (*id_cliente, número_empréstimo*)
depositante = (*id_cliente, número_conta*)
banqueiro_cliente = (*id_cliente, id_funcionário, tipo*)
trabalha_para = (*id_funcionário_trabalhador, id_funcionário_gerente*)

- Esquemas derivados de um conjunto de entidades fraco (lembre-se de que na seção "Redundância de esquemas", provamos que a tabela para *pagamento_empréstimo* é redundante):

pagamento = (*número_empréstimo, número_pagamento, data_pagamento, quantia_pagamento*)

- Esquemas derivados de um relacionamento ISA: (Escolhemos a primeira das duas alternativas apresentadas na Seção "Representação da generalização" de modo a permitir contas que não sejam nem contas de poupança nem contas correntes.)

conta_poupança = (*número_conta, taxa_juros*)
conta_corrente = (*número_conta, quantia_saque_descoberto*)

Deixamos como exercício para você criar restrições de chave estrangeira apropriadas para essas relações.

Outros aspectos do projeto de banco de dados

Nossa longa discussão do projeto de esquema neste capítulo pode criar a falsa impressão de que o projeto de esquema é o único componente de um projeto de banco de dados. Na realidade, existem várias outras considerações que trataremos mais profundamente em capítulos posteriores e descreveremos brevemente aqui.

Restrições de dados e projeto de banco de dados relacional

Vimos uma variedade de restrições de dados que podem ser expressos usando SQL, incluindo restrições de chave primária, restrições de chave estrangeira, restrições de verificação, afirmações e triggers. As restrições servem a vários propósitos. O mais óbvio deles é a automação da preservação de consistência. Expressando restrições na linguagem de definição de dados SQL, o projetista é capaz de assegurar que o próprio sistema de banco de dados imponha as restrições. Isso é mais seguro que confiar em cada programa de aplicação individualmente para impor restrições. Isso também fornece um local central para a atualização das restrições e a adição de novas restrições.

Outra vantagem de declarar restrições de forma explícita é que certas restrições são particularmente úteis no projeto de esquema de banco de dados relacional. Se sabemos, por exemplo, que um número de seguro social identifica unicamente uma pessoa, então podemos usar o número do seguro social de uma pessoa para vincular dados relacionados a essa pessoa, ainda que esses dados apareçam em várias relações. Compare isso, por exemplo, com a cor dos olhos, que não é um identificador único. A cor dos olhos não poderia ser usada para vincular dados pertencentes a uma pessoa específica entre relações, pois os dados dessa pessoa não poderiam ser distinguidos dos dados pertencentes a outras pessoas com a mesma cor de olhos.

Na seção "Redução aos esquemas relacionais", geramos um conjunto de esquemas de relação para um determinado projeto E-R usando as restrições especificadas no projeto. No Capítulo 7, formalizamos essa idéia, bem como idéias relacionadas, e mostramos como ela pode ajudar no projeto do esquema de banco de dados relacional. O método formal para o projeto de banco de dados relacional permite afirmar, de uma maneira precisa, quando um certo projeto é um bom projeto e transformar projetos fracos em projetos melhores. Veremos que o processo de começar com um projeto de relação de entidades e gerar esquemas de relação algoritmicamente desse projeto fornece um bom início para o processo de projeto.

As restrições de dados também são úteis para determinar a estrutura física dos dados. Pode ser útil armazenar dados intimamente relacionados com uma proximidade física no disco, de modo a melhorar a eficiência no acesso ao disco. Certas estruturas de índice funcionam melhor quando o índice está em uma chave primária.

A imposição de restrições ocorre a um preço potencialmente alto no desempenho cada vez que o banco de dados é atualizado. Para cada atualização, o sistema precisa verificar todas as restrições e rejeitar atualizações que falham as restrições ou executar triggers apropriados. A importância do ônus de desempenho depende não só da freqüência de atualização, mas também de como o banco de dados é projetado. Na verdade, a eficiência de testar certos tipos de restrições é um aspecto importante da discussão do esquema de banco de dados relacional no Capítulo 7.

Necessidades do usuário: consultas, desempenho

O desempenho do sistema de banco de dados é um aspecto fundamental da maioria dos sistemas de informação em presariais. O desempenho pertence não apenas ao uso eficiente do hardware de computação e de armazenamento sendo utilizado, mas também da eficiência das pessoas que interagem com o sistema e dos processos que dependem dos dados do banco de dados.

Existem duas métricas principais para o desempenho.

- Vazão – O número de consultas ou atualizações (normalmente chamado de *transações*) que podem ser processados, em média, por unidade do tempo.
- Tempo de resposta – A quantidade de tempo que uma *transação* leva do início ao fim no caso médio ou no pior caso.

Os sistemas que processam altos números de transações em lote dão prioridade à alta vazão. Os sistemas que interagem com pessoas ou os sistemas em que o tempo é um fator crítico normalmente focalizam o tempo de resposta. Essas duas métricas não são equivalentes. A alta vazão surge da obtenção da alta utilização dos componentes do sistema. Isso pode fazer com que certas transações sejam atrasadas até o momento em que possam ser executadas mais eficientemente. Essas transações atrasadas experimentam um tempo de resposta ruim.

A maioria dos sistemas de banco de dados comerciais historicamente tem focalizado a vazão. Entretanto, várias aplicações, incluindo aplicações baseadas na Web e sistemas de informação de telecomunicações, exigem um bom tempo de resposta médio e um limite razoável no tempo de resposta de pior caso.

Um entendimento dos tipos de consultas que são esperados como os mais freqüentes ajuda no processo de proje-

to. Consultas que envolvem junções exigem mais recursos para avaliar que aquelas que não envolvem. Nesses casos, em que uma junção é necessária, o administrador de banco de dados pode preferir criar um índice que facilite a avaliação dessa junção. Para consultas – quer uma junção esteja envolvida ou não –, índices podem ser criados para agilizar a avaliação dos predicados de seleção (cláusula **where** SQL) que provavelmente aparecerão. Outro aspecto das consultas que afeta a escolha de índices é o misto relativo de operações de atualização e de leitura. Embora o índice possa acelerar as consultas, ele também torna as atualizações mais lentas, as quais são forçadas a realizar trabalho extra para manter a precisão do índice.

Necessidades de autorização

As restrições de autorização também afetam o projeto do banco de dados, já que a SQL permite que seja concedido acesso aos usuários na base dos componentes do projeto lógico do banco de dados. Um esquema de relação pode precisar ser decomposto em dois ou mais esquemas para facilitar a concessão de direitos de acesso na SQL. Por exemplo, um registro de funcionario pode incluir dados relativos à folha de pagamento, funções do cargo e benefícios médicos. Como diferentes unidades administrativas da empresa podem gerenciar cada um desses tipos de dados, alguns usuários precisarão acessar os dados de folha de pagamento enquanto terão acesso negado aos dados de cargos, dados médicos etc. Se esses dados estiverem todos em uma tabela, a divisão desejada do acesso, embora ainda possível com o uso de views, será mais complicada. A divisão dos dados dessa maneira se torna ainda mais importante quando os dados são distribuídos entre sistemas em uma rede de computadores, uma questão que consideraremos no Capítulo 22.

Fluxo de dados, fluxo de trabalho

As aplicações de banco de dados normalmente são parte de uma aplicação empresarial mais ampla que interage não apenas com o sistema de banco de dados, mas também com várias aplicações especializadas. Por exemplo, em uma empresa industrial, um sistema de projeto auxiliado por computador (CAD) pode ajudar no projeto de novos produtos. O sistema de CAD pode extrair dados do banco de dados por meio de uma instrução SQL, processar os dados internamente, talvez interagindo com um projetista de produtos, e depois atualizar o banco de dados. Durante esse processo, o controle dos dados pode passar entre vários projetistas de produtos, bem como por outras pessoas. Como outro exemplo, considere um relatório de despesas de viagem. Ele é criado por um funcionário retornando de uma viagem de negócios (possivelmente por meio de um pacote de software especial) e é direcionado subsequentemente para o gerente do funcionário, talvez para outros gerentes superiores e, finalmente, para o departamento de contabilidade para o pagamento (ponto em que ele interage com os sistemas de informação de contabilidade da empresa).

O termo *fluxo de trabalho* se refere à combinação de dados e tarefas envolvidos em processos como os dos exemplos anteriores. Os fluxos de trabalho interagem com o sistema de banco de dados enquanto movem entre usuários e estes realizam suas tarefas no fluxo de trabalho. Além dos dados em que os fluxos de trabalho operam, o banco de dados pode armazenar dados sobre o próprio fluxo de trabalho, incluindo as tarefas compondo um fluxo de trabalho e como elas devem ser direcionadas entre usuários. Os fluxos de trabalho, portanto, especificam uma série de consultas e atualizações no banco de dados que podem ser levadas em conta como parte do processo de projeto de banco de dados. Em outras palavras, modelar a empresa exige não só entender a semântica dos dados, mas também os processos empresariais que usam esses dados.

Outras questões do projeto de banco de dados

O projeto de banco de dados normalmente não é uma atividade de uma única vez. As necessidades de uma organização estão continuamente evoluindo, e os dados que ela precisa armazenar também crescem proporcionalmente. Durante as fases iniciais do projeto de banco de dados, ou durante o desenvolvimento de uma aplicação, o projetista de banco de dados pode perceber que é necessário efetuar mudanças nos níveis de esquema conceitual, lógico ou físico. As mudanças no esquema podem afetar todos os aspectos da aplicação de banco de dados. Um bom projeto de banco de dados prevê as futuras necessidades de uma organização e projeta o esquema de uma maneira que mudanças mínimas sejam necessárias à medida que aumentam as necessidades.

É importante distinguir entre restrições fundamentais e restrições que têm previsão de mudança. Por exemplo, a restrição de que um id_cliente identifica um cliente único é fundamental. Por outro lado, um banco pode ter uma política de que um cliente só pode ter uma conta, que pode mudar futuramente. Um projeto de banco de dados que apenas permite uma conta por cliente necessitaria de grandes mudanças se o banco de dados mudasse sua política. Essas mudanças não devem exigir uma grande alteração no projeto de banco de dados.

Além disso, a empresa que o banco de dados está servindo provavelmente interage com outras empresas e, portanto, vários bancos de dados podem precisar interagir. A conversão dos dados entre diferentes esquemas é um problema importante nas aplicações reais. Várias soluções têm sido propostas

para esse problema. O modelo de dados XML, que estudaremos no Capítulo 10, é amplamente usado para representar dados quando são trocados entre diferentes aplicações.

Finalmente, vale notar que o projeto de banco de dados é uma atividade orientada para humanos em dois sentidos: os usuários finais do sistema são pessoas (mesmo que haja uma aplicação entre o banco de dados e os usuários finais); e o projetista de banco de dados precisa interagir extensivamente com especialistas no domínio da aplicação para entender as necessidades de dados da aplicação. Todas as pessoas envolvidas com os dados possuem necessidades e preferências que devem ser levadas em conta a fim de que o projeto e a utilização do banco de dados tenham sucesso dentro da empresa.

A Unified Modeling Language (UML)**

Os diagramas de entidade-relacionamento ajudam a modelar o componente de representação de dados de um sistema de software. A representação de dados, contudo, forma apenas uma parte de um projeto de sistema geral. Outros componentes incluem modelos das interações do usuário com o sistema, especificação de módulos funcionais do sistema e sua interação etc. A Unified Modeling Language (UML) é um padrão desenvolvido sob o patrocínio do Object Management Group (OMG) para criar especificações de vários componentes de um sistema de software. Algumas partes da UML são:

- **Diagrama de classe**. Um diagrama de classe é semelhante a um diagrama E-R. Mais adiante nesta seção, ilustraremos alguns recursos dos diagramas de classe e como eles se relacionam aos diagramas E-R.
- **Diagrama de caso de uso**. Os diagramas de caso de uso mostram a interação entre os usuários e o sistema, em especial, as etapas das tarefas que os usuários realizam (como sacar dinheiro ou se matricular em um curso).
- **Diagrama de atividade**. Os diagramas de atividade representam o fluxo das tarefas entre vários componentes de um sistema.
- **Diagrama de implementação**. Os diagramas de implementação mostram os componentes de sistema e suas interconexões, tanto em nível de componente de software quanto de componente de hardware.

Não tentamos fornecer cobertura detalhada das diferentes partes da UML aqui. Veja as notas bibliográficas para obter referências sobre a UML. Entretanto, ilustramos com exemplos alguns recursos da parte da UML relacionada à modelagem de dados.

A Figura 6.28 mostra várias construções de diagrama E-R e suas construções de diagrama de classe UML equivalentes. Descrevemos essas construções a seguir. A UML mostra conjuntos de entidades como retângulos e, diferente do diagrama E-R, mostra atributos dentro do retângulo e não como elipses separadas. Na verdade, a UML modela objetos, enquanto o E-R modela entidades. Os objetos são como entidades e possuem atributos, mas, além disso, fornecem um conjunto de funções (chamadas métodos) que podem ser chamadas para calcular valores na base dos atributos dos objetos, ou para atualizar o próprio objeto. Os diagramas de classe podem descrever métodos e atributos. Abordaremos os objetos no Capítulo 9.

Representamos conjuntos de relacionamentos binários na UML simplesmente desenhando uma linha conectando os conjuntos de entidades. Escrevemos o nome do conjunto de relacionamentos adjacente à linha. Também podemos especificar o papel desempenhado por um conjunto de entidades em um conjunto de relacionamentos escrevendo o nome do papel na linha, adjacente ao conjunto de entidades. Como alternativa, podemos escrever o nome do conjunto de relacionamentos em um retângulo, juntamente com atributos do conjunto de relacionamento, e conectar o retângulo por uma linha pontilhada até a linha representando o conjunto de relacionamento. Esse retângulo, então, pode ser tratado como um conjunto de entidades, da mesma maneira que uma agregação nos diagramas E-R, e pode participar em relacionamentos com outros conjuntos de entidades.

Desde a versão 1.3, a UML aceita relacionamentos não binários, usando a mesma notação de losango dos diagramas E-R. Os relacionamentos não binários não podiam ser representados diretamente nas versões anteriores da UML – eles tinham de ser convertidos em relacionamentos binários pela técnica que vimos anteriormente na seção "Conjuntos de relacionamento binários *versus* enários".

As restrições de cardinalidade são especificadas na UML da mesma maneira que nos diagramas E-R, na forma $l..h$, onde l indica o mínimo e h, o número máximo de relacionamentos em que um conjunto de entidades pode participar. Entretanto, você deve estar atento ao fato de que o posicionamento das restrições é exatamente o inverso do posicionamento das restrições nos diagramas E-R, como mostra a Figura 6.28. A restrição 0..* no lado $E2$ e a restrição 0..1 no lado $E1$ significam que cada entidade $E1$ pode participar em muitos relacionamentos; ou seja, o relacionamento é muitos-para-um de $E2$ para $E1$.

Valores únicos como 1 ou * podem ser escritos nas bordas; o valor único 1 em uma borda é tratado como equivalente a 1..1, enquanto * é equivalente a 0..*.

Representamos a generalização e a especialização na UML conectando conjuntos de entidades por uma linha com um triângulo na ponta correspondente ao conjunto de entidades mais geral. Por exemplo, o conjunto de entidades

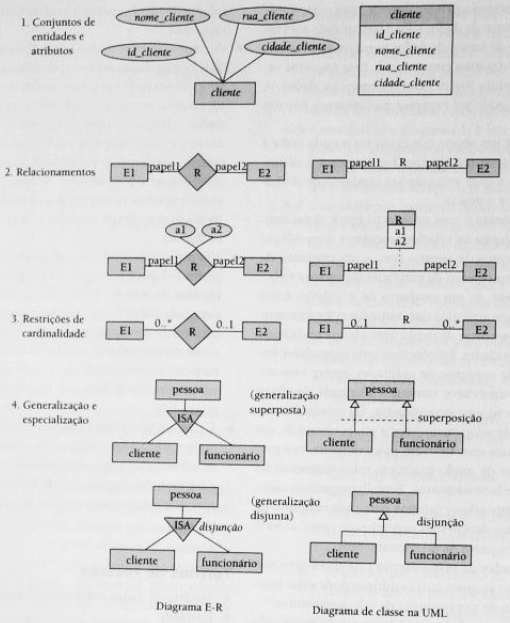


Figura 6.28 Símbolos usados na notação do diagrama de classe UML.

pessoa é uma generalização de *cliente* e *funcionário*. Os diagramas UML também podem representar explicitamente as restrições de disjunção/superposição nas generalizações. A Figura 6.28 mostra generalizações disjuntas e superpostas de *cliente* e *funcionário* para *pessoa*. Lembre-se de que se a generalização *cliente*/*funcionário* para *pessoa* for disjunta, isso significa que ninguém pode ser um *cliente* e um *funcionário*. Uma generalização superposta permite que uma pessoa seja um *cliente* e um *funcionário*.

Os diagramas de classe UML incluem várias outras notações que não correspondem às notações E-R que vimos. Por exemplo, uma linha entre dois conjuntos de entidades com um losango em uma ponta especifica que a entidade no lado do losango contém a outra entidade (a contenção é chamada de "agregação" na terminologia UML). Por exemplo, uma entidade veículo pode conter uma entidade motor. Os diagramas de classe UML também fornecem notações para representar recursos de linguagem orientados a objeto, como anotações públicas ou privadas dos membros de classe, e interfaces (estas devem ser familiares a qualquer um que conheça a linguagem Java ou C#). Veja as referências nas notas bibliográficas para obter mais informações sobre diagramas de classe UML.

Resumo

- O projeto de banco de dados envolve principalmente o projeto do esquema de banco de dados. O modelo de dados **entidade-relacionamento (E-R)** é um modelo de dados amplamente usado para projeto de banco de dados. Ele fornece uma representação gráfica conveniente para ver dados, relacionamentos e restrições.

- O modelo é destinado principalmente para o processo de projeto de banco de dados e foi desenvolvido para facilitar o projeto de banco de dados permitindo a especificação de um **esquema empresarial**. Esse esquema representa a estrutura lógica geral do banco de dados. A estrutura geral pode ser expressa graficamente por um **diagrama E-R**.
- Uma **entidade** é um objeto que existe no mundo real e é distinguível dos outros objetos. Expressamos a distinção associando a cada entidade um conjunto de atributos que descreva o objeto.
- Um **relacionamento** é uma associação entre várias entidades. Um **conjunto de relacionamentos** é uma coleção de relacionamentos do mesmo tipo, e um **conjunto de entidades** é uma coleção de entidades do mesmo tipo.
- Uma **superchave** de um conjunto de entidades é um conjunto ou mais atributos que, tomados coletivamente, nos permite identificar de modo único uma entidade no conjunto de entidades. Escolhemos uma superchave mínima para cada conjunto de entidades dentre suas superchaves; a superchave mínima é chamada de **chave primária** do conjunto de entidades. Da mesma forma, um conjunto de relacionamentos é um conjunto de um ou mais atributos que, tomados coletivamente, nos permite identificar de modo único um relacionamento no conjunto de relacionamentos. Semelhantemente, escolhemos uma superchave mínima para cada conjunto de relacionamentos dentre suas superchaves; essa é a chave primária do conjunto de relacionamentos.
- As **cardinalidades de mapeamento** expressam o número de entidades às quais outra entidade pode estar associada por meio de um conjunto de relacionamentos.
- Um conjunto de entidades que não possui atributos suficientes para formar uma chave primária é considerado um **conjunto de entidades fraco**. Um conjunto de entidades que possui uma chave primária é considerado um **conjunto de entidades forte**.
- **Especialização** e **generalização** definem um relacionamento de contenção entre um conjunto de entidades de nível superior e um ou mais conjuntos de entidades de nível inferior. Especialização é o resultado de tomar um subconjunto de um conjunto de entidades de nível superior para formar um conjunto de entidades de nível inferior. Generalização é o resultado de tomar a união de dois ou mais conjuntos de entidades disjuntos (de nível inferior) para produzir um conjunto de entidades de nível superior. Os atributos dos conjuntos de entidades de nível superior são herdados pelos conjuntos de entidades de nível inferior.
- **Agregação** é uma abstração em que os conjuntos de relacionamentos (juntamente com seus conjuntos de entidades associados) são tratados como conjuntos de entidades de nível superior e podem participar em relacionamentos.
- Os vários recursos do modelo E-R oferecem ao projetista de banco de dados inúmeras escolhas de como representar melhor a empresa sendo modelada. Conceitos e objetos podem, em certos casos, ser representados por entidades, relacionamentos ou atributos. Os aspectos da estrutura geral da empresa podem ser mais bem descritos usando conjuntos de entidades fracos, generalização, especialização ou agregação. Muitas vezes, o projetista precisa pesar as vantagens de um modelo simples e compacto com as de um modelo mais preciso, porém, mais complexo.
- Um projeto de banco de dados especificado por um diagrama E-R pode ser representado por uma coleção de esquemas de relação. Para cada conjunto de entidades e para cada conjunto de relacionamentos no banco de dados, existe um esquema de relação único que recebe o nome do conjunto de entidades ou conjunto de relacionamentos correspondente. Isso forma a base para derivar um projeto de banco de dados relacional de um diagrama E-R.
- A **Unified Modeling Language (UML)** fornece um meio gráfico de modelar vários componentes de um sistema de software. O componente diagrama de classe da UML é baseado em diagramas E-R. Entretanto, existem algumas diferenças entre os dois de que precisamos estar cientes.

Termos de revisão

- Modelo de dados entidade-relacionamento
- Entidade
- Conjunto de entidades
- Relacionamento e conjunto de relacionamentos
- Papel
- Conjunto de relacionamentos recursivo
- Atributos descritivos
- Conjunto de relacionamentos binário
- Grau do conjunto de relacionamentos
- Atributos
- Domínio
- Atributos simples e compostos
- Atributos de valor único e valores múltiplos
- Valor nulo
- Atributo derivado
- Superchave, chave candidata e chave primária
- Cardinalidade de mapeamento:
  - Relacionamento um-para-um
  - Relacionamento um-para-muitos
  - Relacionamento muitos-para-um
  - Relacionamento muitos-para-muitos

- Participação
  - Participação total
  - Particionamento parcial
- Conjuntos de entidades fracos e conjuntos de entidades fortes
  - Atributos discriminadores
  - Relacionamento de identificação
- Especialização e generalização
  - Superclasse e subclasse
  - Herança de atributo
  - Herança única e múltipla
  - Membros definidos por condição e pelo usuário
  - Disjunção e generalização superposta
- Restrição de integralidade
  - Generalização total e parcial
- Agregação
- Diagrama E-R
- Unified Modeling Language (UML)

Exercícios práticos

6.1 Construa um diagrama E-R para uma seguradora de automóveis em que cada cliente possui um ou mais carros. Cada carro tem associado a ele zero a qualquer número de acidentes registrados.

6.2 Um órgão de registro universitário mantém dados sobre as seguintes entidades: (a) cursos, incluindo número, título, créditos, roteiro e pré-requisitos; (b) ofertas de cursos, incluindo número do curso, ano, semestre, número de seção, instrutor(es), programações e turma; (c) alunos, incluindo id_aluno, nome e programa; e (d) instrutores, incluindo número de identificação, nome, departamento e título. Além disso, a matrícula de alunos em cursos e os períodos cumpridos pelos alunos em cada curso em que são matriculados precisam ser corretamente modelados.

Construa um diagrama E-R para o órgão de registro. Documente todas as suposições que você fizer sobre as restrições de mapeamento.

6.3 Considere um banco de dados usado para registrar as notas que os alunos tiram em diferentes exames de diferentes cursos.

a. Construa um diagrama E-R que modele exames como entidades e use um relacionamento ternário para o banco de dados.

b. Construa um diagrama E-R alternativo que use apenas um relacionamento binário entre *alunos* e *ofertas_curso*. Certifique-se de que apenas um relacionamento exista entre um determinado par aluno e oferta de curso, embora você possa representar as notas que um aluno tira em diferentes exames de um curso.

6.4 Projete um diagrama E-R para controlar as campanhas do seu time de esporte favorito. Você deve armazenar os jogos realizados, os pontos em cada jogo, os jogadores em cada partida e as estatísticas de jogadores individuais em cada partida. Estatísticas gerais devem ser modeladas como atributos derivados.

6.5 Considere um diagrama E-R em que o mesmo conjunto de entidades apareça várias vezes. Por que permitir essa redundância é uma má prática que deve ser evitada sempre que possível?

6.6 Considere um banco de dados universitário para a programação das turmas para exames finais. Esse banco de dados deve ser modelado como o conjunto de entidades único *exame*, com atributos *nome_curso*, *número_seção*, *número_sala* e *hora*. Alternativamente, um ou mais conjuntos de entidades adicionais podem ser definidos, juntamente com conjuntos de relacionamentos para substituir alguns dos atributos do conjunto de entidades *exame*, como

- *curso*, com atributos *nome*, *departamento* e *número_turma*
- *seção*, com atributos *número_seção* e *matrícula*, e dependente como um conjunto de entidades fraco em *curso*
- *sala*, com atributos *número_sala*, *capacidade* e *prédio*

a. Mostre um diagrama E-R ilustrando o uso de todos os três conjuntos de entidades adicionais listados.

b. Explique que características de aplicação influenciariam em uma decisão de incluir ou não cada um dos conjuntos de entidades adicionais.

6.7 Ao projetar um diagrama E-R para uma determinada empresa, você tem várias alternativas dentre as quais escolher.

a. Que critérios você deve considerar para fazer a escolha certa?

b. Projete três diagramas E-R alternativos para representar o órgão de registro universitário do Exercício prático 6.2. Liste as vantagens de cada um. Argumente a favor de uma das alternativas.

6.8 Um diagrama E-R pode ser visto como um gráfico. O que os seguintes itens significam em termos da estrutura de um esquema empresarial?

a. O gráfico está desconectado.

b. O gráfico é acíclico.

6.9 Considere a representação de um relacionamento ternário usando relacionamentos binários como descrito na seção "Conjuntos de relacionamento binários *versus* enários" e ilustrado na Figura 6.29 (atributos não-mostrados).

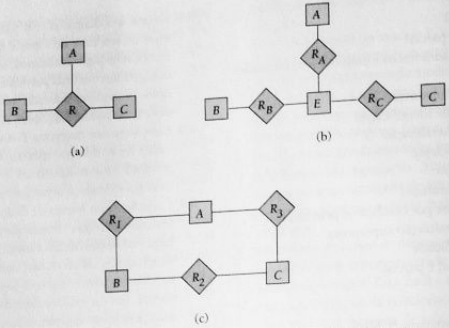


Figura 6.29 Diagrama E-R para os Exercícios práticos 6.9 e 6.22.

a. Mostre uma instância simples de E, A, B, C, R_A, R_B e R_C que não pode corresponder a qualquer instância de A, B, C e R.
b. Modifique o diagrama E-R da Figura 6.29b para introduzir restrições que garantirão que qualquer instância de E, A, B, C, R_A, R_B e R_C que satisfaça as restrições irá corresponder a uma instância de A, B, C e R.
c. Modifique essa tradução para manipular as restrições de participação total no relacionamento ternário.
d. A representação anterior exige que criemos um atributo de chave primária para E. Mostre como tratar E como um conjunto de entidades fraco de modo que um atributo de chave primária não seja necessário.

6.10 Um conjunto de entidades fraco sempre pode ser transformado em um conjunto de entidades forte acrescentando a seus atributos os atributos de chave primária do seu conjunto de entidades identificador. Descreva que tipo de redundância resultará se fizermos isso.

6.11 A Figura 6.30 mostra uma estrutura em treliça da generalização e especialização (atributos não mostrados). Para os conjuntos de entidades A, B e C, explique como os atributos são herdados dos conjuntos de entidades de nível superior X e Y. Discuta como tratar um caso em que um atributo de X possui o mesmo nome de algum atributo de Y.

6.12 Considere dois bancos separados que decidem se fundir. Suponha que ambos os bancos usam exatamente o mesmo esquema de banco de dados E-R – o representado na Figura 6.25. (É claro, essa suposição é altamente irrealista; consideramos o caso mais realista na seção "Bancos de dados distribuídos heterogêneos" do Capítulo 22.) Se o banco, após a fusão, precisar ter um único banco de dados, existirão vários problemas em potencial:

- A possibilidade de que os dois bancos originais tenham agências com o mesmo nome
- A possibilidade de que alguns clientes sejam clientes dos dois bancos originais
- A possibilidade de que alguns números de empréstimo ou de conta fossem usados nos dois bancos originais (para diferentes empréstimos ou contas, é claro)

Para cada um desses problemas potenciais, descreva por que existe realmente um potencial para dificuldades. Proponha uma solução para o problema. Em sua solução, explique quaisquer mudanças que precisariam ser feitas e descreva qual seria seu efeito sobre o esquema e os dados.

6.13 Reconsidere a situação descrita para o Exercício prático 6.12 sob a suposição de que um banco está nos Estados Unidos e o outro, no Canadá. Como antes, os bancos usam o esquema da Figura 6.25, exceto que o banco canadense usa o número do "*social_insurance*" atribuído pelo governo canadense, enquanto o banco americano usa o número do "*social_security*" para identificar clientes. Que problemas (além dos identificados no Exercício prático 6.11) poderiam ocorrer nesse caso multinacional? Como você resolveria esse problema? Certifique-se de considerar os dois esquemas e os valores de dados reais na construção de sua resposta.

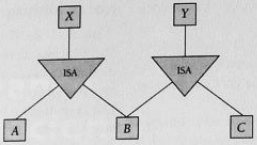


Figura 6.30 Diagrama E-R para o Exercício prático 6.11.

Exercícios

6.14 Explique as diferenças entre os termos chave primária, chave candidata e superchave.

6.15 Construa um diagrama E-R para um hospital com um conjunto de pacientes e um conjunto de médicos. Associe a cada paciente um log dos vários testes e exames realizados.

6.16 Construa tabelas apropriadas para cada um dos diagramas nos Exercícios práticos 6.1 a 6.2.

6.17 Estenda o diagrama E-R do Exercício prático 6.4 para rastrear as mesmas informações para todos os times de um campeonato.

6.18 Explique a diferença entre um conjunto de entidades fraco e um conjunto de entidades forte.

6.19 Podemos converter qualquer conjunto de entidades fraco em um conjunto de entidades forte simplesmente acrescentando atributos apropriados. Por que, então, temos conjuntos de entidades fracos?

6.20 Defina o conceito de agregação. Forneça dois exemplos em que esse conceito é útil.

6.21 Considere o diagrama E-R da Figura 6.31, que modela uma livraria on-line.

a. Liste os conjuntos de entidades e suas chaves primárias.

b. Suponha que a livraria adicione fitas cassete e CDs de música à sua coleção. A mesma gravação pode estar presente no formato cassete ou CD, com preços diferentes. Estenda o diagrama

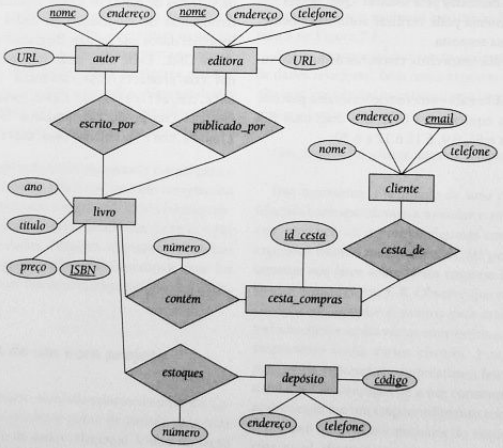


Figura 6.31 Diagrama E-R para o Exercício 6.21.

E-R para modelar essa adição, ignorando o efeito nas cestas de compras.

c. Agora estenda o diagrama E-R, usando generalização, para modelar o caso em que uma cesta de compras pode conter qualquer combinação de livros, cassetes ou CDs.

6.22 Na seção "Conjuntos de relacionamento binários *versus* enários", representamos um relacionamento ternário (repetido na Figura 6.29a) usando relacionamentos binários, como mostra a Figura 6.29b. Considere a alternativa mostrada na Figura 6.29c. Discuta as vantagens relativas dessas duas representações alternativas de um relacionamento ternário por relacionamentos binários.

6.23 Considere os esquemas de relação mostrados na seção "Esquemas relacionais para instituição bancária", que são gerados do diagrama E-R na Figura 6.25. Para cada esquema, especifique que restrições de chave estrangeira, se houver, devem ser criadas.

6.24 Projete uma hierarquia generalização-especialização para uma revendedora de veículos automotores. A empresa vende motocicletas, carros de passeio, vans e ônibus. Justifique seu posicionamento de atributos em cada nível da hierarquia. Explique por que eles não devem ser colocados em um nível mais alto ou mais baixo.

6.25 Explique a diferença entre restrições definidas por condição e definidas pelo usuário. Qual dessas restrições o sistema pode verificar automaticamente? Explique sua resposta.

6.26 Explique a diferença entre restrições disjuntas e superpostas.

6.27 Explique a diferença entre restrições totais e parciais.

6.28 Desenhe os equivalentes UML dos diagramas E-R das Figuras 6.8c, 6.9, 6.11 6.12 e 6.20.

Notas bibliográficas

O modelo de dados E-R foi introduzido por Chen [1976]. Uma metodologia de projeto lógica para bancos de dados relacionais usando o modelo E-R estendido é apresentada por Teorey *et al.* [1986]. O padrão Integration Definition for Information Modeling (IDEF1X) [1993], lançado pelo United States National Institute of Standards and Technology (NIST) definiu padrões para diagramas E-R. Entretanto, diversas notações E-R estão em uso atualmente.

Thalheim [2000] fornece um livro-texto detalhado que abrange a pesquisa na modelagem E-R. Livros-texto de discussões básicas são oferecidos por Batini *et al.* [1992] e Elmasri e Navathe [2003]. Davis *et al.* [1983] fornecem uma coleção de documentos sobre o modelo E-R.

Em 2004, a versão da UML era a 1.5. Veja `www.uml.org` para obter mais informações sobre padrões e ferramentas da UML 2.0.

Ferramentas

Muitos sistemas de banco de dados fornecem ferramentas para projeto de banco de dados que aceitam diagramas E-R. Essas ferramentas ajudam o projetista a criar diagramas E-R e podem criar automaticamente tabelas correspondentes em um banco de dados. Veja as notas bibliográficas do Capítulo 1 para ter referências dos sites dos fornecedores de sistemas de banco de dados. Existem também algumas ferramentas de modelagem de dados independentes de banco de dados que aceitam diagramas E-R e diagramas de classe UML. Entre elas, estão Rational Rose (`www.rational.com/products/rose`, Microsoft Visio (`www.microsoft.com/office/visio`), ERwin (pesquise ERwin no site `www.cai.com/products`), Poseidon for UML (`www.gentleware.com` e SmartDraw (`www.smartdraw.com`).

CAPÍTULO 7

Projeto de banco de dados relacional

Neste capítulo, consideramos o problema de projetar um esquema para um banco de dados relacional. Muitos dos problemas em fazer isso são semelhantes aos problemas de projeto que consideramos no Capítulo 6 usando o modelo E-R.

Em geral, o objetivo do projeto de banco de dados relacional é um conjunto de esquemas de relação que nos permite armazenar informações sem redundância desnecessária, além de permitir recuperar informações facilmente. Isso é conseguido projetando esquemas que estejam em uma *forma normal* apropriada. Para determinar se um esquema de relação está em uma das formas normais desejáveis, precisamos de informações sobre a empresa real que estamos modelando com o banco de dados. Algumas dessas informações existem em um diagrama E-R bem desenhado, mas também podem ser necessárias informações adicionais sobre a empresa.

Neste capítulo, apresentamos um método formal para o projeto de banco de dados relacional baseado no conceito de dependências funcionais. Depois, definimos formas normais em termos de dependências funcionais e outros tipos de dependências de dados. Primeiro, entretanto, examinamos o problema do projeto relacional do ponto de vista dos esquemas derivados de um determinado projeto de entidade-relacionamento.

Características de um bom projeto relacional

Nosso estudo do projeto entidade-relacionamento no Capítulo 6 fornece um excelente ponto de partida para criar um projeto de banco de dados relacional. Vimos na seção "Redução aos esquemas relacionais" do Capítulo 6 que é possível gerar um conjunto de esquemas de relação diretamente do projeto E-R. Obviamente, a qualidade do conjunto de esquemas resultante depende da qualidade do projeto E-R tinha em primeiro lugar. Mais adiante neste capítulo, estudaremos meios precisos de avaliar se uma coleção de esquemas de relação é desejável ou não. Entretanto, podemos dar um grande passo em direção a um bom projeto usando conceitos que já estudamos.

Para facilitar a consulta, repetimos os esquemas da seção "Esquemas relacionais para instituição bancária" do Capítulo 6 na Figura 7.1.

Agora, vamos explorar recursos desse projeto de banco de dados relacional, bem como algumas alternativas. Suponha que, em vez dos esquemas *tomador* e *empréstimo*, tivéssemos o esquema:

tom_empr = (*id_cliente*, *número_empréstimo*, *quantia*)

Isso representa o resultado de uma junção natural nas relações correspondentes a *tomador* e *empréstimo*. Isso parece uma boa idéia porque algumas consultas podem ser expressas usando menos junções, até pensarmos cuidadosamente nos fatos sobre nossa empresa bancária que levaram ao nosso projeto E-R. Observe que o conjunto de relacionamento *tomador* é muitos-para-muitos. Isso permite que um cliente tenha vários empréstimos e também que um empréstimo tenha vários clientes. Fizemos essa escolha para poder representar empréstimos feitos conjuntamente a um casal de cônjuges ou a um consórcio de pessoas (que podem estar em um empreendimento comercial conjunto). É por isso que a chave primária do esquema *tomador* consiste no *id_cliente* e no *número_empréstimo* em vez de apenas no *número_empréstimo*.

agência = (*nome_agência*, *cidade_agência*, *ativo*)
cliente = (*id_cliente*, *nome_cliente*, *rua_cliente*, *cidade_cliente*)
empréstimo = (*número_empréstimo*, *quantia*)
conta = (*número_conta*, *saldo*)
funcionário = (*id_funcionário*, *nome_funcionário*, *número_telefone*, *data_admissão*)
nome_dependente = (*id_funcionário*, *nome_dependente*)
agência_conta = (*número_conta*, *nome_agência*)
agência_empréstimo = (*número_empréstimo*, *nome_agência*)
tomador = (*id_cliente*, *número_empréstimo*)
depositante = (*id_cliente*, *número_conta*)
banqueiro_cliente = (*id_cliente*, *id_funcionário*, *tipo*)
trabalha_para = (*id_funcionário_trabalhador*, *id_funcionário_gerente*)
pagamento = (*número_empréstimo*, *número_pagamento*, *data_pagamento*, *quantia_pagamento*)
conta_poupança = (*número_conta*, *taxa_juros*)
conta_corrente = (*número_conta*, *quantia_saque_descoberto*)

Figura 7.1 Os esquemas de banco da seção "Esquemas relacionais para instituição bancária" do Capítulo 6.

Vamos considerar um empréstimo que é feito a um consórcio e considerar as tuplas que precisam estar na relação no esquema *tom_empr*. Suponha que o número de empréstimo L-100 seja feito a um consórcio consistindo nos seguintes clientes: James (com id_cliente 23-652), Anthony (com id_cliente 15-202) e Jordan (com id_cliente 23-521) na quantia de 10.000 dólares.

A Figura 7.2 mostra como isso seria representado usando *empréstimo* e *tomador* e como seria representado no projeto alternativo usando *tom_empr*. A tupla (L-100, 10000) na relação no esquema *empréstimo* se junta a três tuplas na relação no esquema *tomador*, gerando três tuplas na relação no esquema *tom_empr*. Observe que em *tom_empr* tivemos de repetir a quantia do empréstimo uma vez para cada cliente no consórcio de pessoas que tomaram o empréstimo. É importante que todas essas tuplas concordem quanto à quantia do empréstimo L-100, já que, caso contrário, nosso banco de dados seria inconsistente. Em nosso projeto original usando *empréstimo* e *tomador*, armazenamos a quantia de cada empréstimo exatamente uma vez. Isso sugere que usar *tom_empr* é uma má idéia, já que ele armazena quantias de empréstimo de ma-

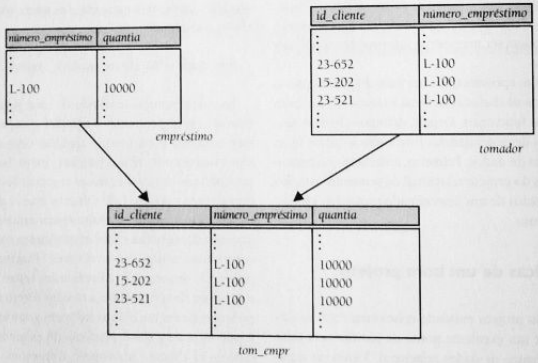


Figura 7.2 Lista parcial das tuplas nas relações *empréstimo*, *tomador* e *tom_empr*.

neira redundante e corre o risco de que algum usuário possa atualizar a quantia de empréstimo em uma tupla mas não em todas e, assim, criar inconsistência.

Agora vamos considerar outra alternativa. Seja *empr_qt_ag* = (*número_empréstimo, quantia, nome_agência*) criado a partir de *agência_empréstimo* e *empréstimo* (por uma junção das relações correspondentes). Isso parece semelhante ao exemplo que acabamos de considerar, mas com uma importante diferença. Aqui, *número_empréstimo* é a chave primária dos dois esquemas *agência_empréstimo* e *empréstimo* e, portanto, também é a chave primária de *empr_qt_ag*. Isso derivou do fato de que o conjunto de relacionamento *agência_empréstimo* é muitos-para-um, diferente do conjunto de relacionamento *tomador* em nosso exemplo anterior. Para um determinado empréstimo, existe apenas uma agência associada. Vamos supor que o empréstimo L-100 esteja associado à agência Springfield. A Figura 7.3 mostra como isso seria representado usando *empr_qt_ag*. A tupla (L-100, 10000) na relação no esquema *empréstimo* junta com apenas uma tupla na relação no esquema *agência_empréstimo*, gerando apenas uma tupla na relação no esquema *empr_qt_ag*. Não há qualquer repetição de informações em *empr_qt_ag* e, portanto, isso evita os problemas que encontramos em nosso exemplo anterior.

Antes de finalmente concordarmos em usar *empr_qt_ag* no lugar de *empréstimo* e *agência_empréstimo*, existe mais uma questão a ser considerada. Poderíamos querer registrar um empréstimo e sua agência associada no banco de dados antes que sua quantia tivesse sido determinada? No projeto antigo, o esquema *agência_empréstimo* pode cuidar disso, mas, no projeto revisado usando *empr_qt_ag*, precisaríamos criar uma tupla com um valor nulo para *quantia*. Em alguns casos, os valores nulos são problemáticos, como vimos anteriormente em nosso estudo da SQL. Entretanto, se decidirmos que isso não é um problema nesse caso, então, podemos continuar a usar o projeto revisado.

Os dois exemplos que acabamos de considerar mostram a importância da natureza das chaves primárias em determinar se a combinação de esquemas faz sentido. Surgiram problemas – especificamente repetição de informações – quando o atributo de junção (*número_empréstimo*) não era a chave primária para *ambos* os esquemas sendo combinados.

Alternativa de projeto: esquemas menores

Suponha novamente que, de algum modo, tivéssemos começado com o esquema *tom_empr*. Como reconheceríamos que ele exige repetição de informações e deve ser desmembrado nos dois esquemas *tomador* e *empréstimo*? Como não teríamos os esquemas *tomador* e *empréstimo*, não teríamos as informações de chave primária que usamos para descrever o problema com *tom_empr*.

Observando o conteúdo de relações reais no esquema *tom_empr*, podemos notar a repetição de informações resultando da necessidade de listar a quantia de empréstimo uma vez para cada tomador associado a um empréstimo. Entretanto, esse é um processo indesejável. Um banco de dados real possui um grande número de esquemas e números ainda maiores de atributos. O número de tuplas pode estar na casa dos milhões ou mais. Descobrir repetição seria oneroso. Há um problema ainda mais crítico com esse método. Ele não permite determinar se a falta de repetição é apenas um caso especial de "sorte" ou se é uma manifestação de uma regra geral. Em nosso exemplo, como *saberíamos* que em nossa instituição bancária cada empréstimo

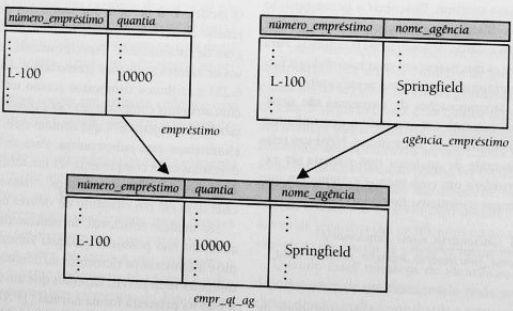


Figura 7.3 Lista parcial das tuplas nas relações *empréstimo*, *agência_empréstimo* e *empr_qt_ag*.

(identificado por seu número de empréstimo) *precisa* ter apenas a quantia? O fato de o número de empréstimo L-100 aparecer três vezes com a mesma quantia é apenas uma coincidência? Não podemos responder a essas perguntas sem voltar à empresa propriamente dita e entender suas regras. Em especial, precisaríamos descobrir se o banco exige que cada empréstimo (identificado por seu número de empréstimo) *precisa* ter apenas uma quantia.

No caso de *tom_empr*, nosso processo de criar um projeto E-R bem-sucedido evitou a criação desse esquema. No entanto, essa situação fortuita nem sempre ocorre. Então, precisamos permitir que o projetista especifique regras como "cada valor específico para *número_empréstimo* corresponde no máximo a uma *quantia*", mesmo em casos em que *número_empréstimo* não é a chave primária para o esquema em questão. Em outras palavras, precisamos escrever uma regra que diga "se houvesse um esquema (*número_empréstimo, quantia*), então, *número_empréstimo* é capaz de agir como a chave primária". Essa regra é especificada como uma **dependência funcional**

número_empréstimo → *quantia*

Dada tal regra, agora temos informações suficientes para reconhecer o problema do esquema *tom_empr*. Como *número_empréstimo* não pode ser a chave primária para *tom_empr* (porque um empréstimo pode precisar de várias tuplas na relação no esquema *tom_empr*), a quantia de um empréstimo pode precisar ser repetida.

Observações como essas, e as regras (dependências funcionais específicas) que delas resultam, permitem que o projetista de banco de dados reconheça situações em que um esquema precisa ser dividido, ou decomposto, em dois ou mais esquemas. Não é difícil ver que a maneira correta de decompor *tom_empr* é nos esquemas *tomador* e *empréstimo*, como no projeto original. Descobrir a decomposição certa é muito mais difícil para esquemas com um grande número de atributos e várias dependências funcionais. Para lidar com isso, iremos nos basear em uma metodologia formal que desenvolveremos mais adiante neste capítulo.

Nem todas as decomposições de esquemas são úteis. Considere um caso extremo em que tudo o que tínhamos eram esquemas consistindo em um atributo. Nenhum relacionamento interessante de qualquer tipo poderia ser expresso. Agora, considere um caso menos extremo em que escolhemos decompor o esquema *funcionário* em

funcionário1 = (*id_funcionário, nome_funcionário*)
funcionário2 = (*nome_funcionário, número_telefone, data_admissão*)

A falha nessa decomposição surge da possibilidade de que a empresa tenha dois funcionários com o mesmo nome. Isso não é improvável na prática, já que muitas culturas possuem certos nomes altamente comuns e, além disso, crianças podem receber nomes dos pais. É claro, cada pessoa teria um id_funcionário único, que é o motivo pelo qual *id_funcionário* pode agir como uma chave primária. Como um exemplo, vamos considerar que dois funcionários, ambos chamados Kim, trabalham para o banco e têm as seguintes tuplas na relação no esquema *funcionário* no projeto original:

(123-45-6789, Kim, 882-0000, 1984-03-29)
(987-65-4321, Kim, 869-9999, 1981-01-16)

A Figura 7.4 mostra essas tuplas, as tuplas resultantes usando os esquemas resultantes da decomposição e o resultado se tentássemos gerar novamente as tuplas originais usando uma junção natural. Como vemos na figura, as duas tuplas originais aparecem no resultado juntamente com duas novas tuplas que misturam incorretamente valores de dados pertencentes aos dois funcionários chamados Kim. Embora tenhamos mais tuplas, na verdade, temos menos informações no seguinte sentido. Podemos indicar que um certo número de telefone e uma certa data de admissão pertencem a alguém chamado Kim, mas somos incapazes de distinguir quem. Portanto, nossa decomposição é incapaz de representar certos fatos importantes sobre a instituição bancária. Claramente, gostaríamos de evitar essas decomposições. Vamos nos referir a essas decomposições como **decomposições com perda** e, ao contrário, àquelas decomposições que não são incapazes como **decomposições sem perda**.

Domínios atômicos e primeira forma normal

O modelo E-R permite que conjuntos de entidades e conjuntos de relacionamento tenham atributos com algum grau de subestrutura. Especificamente, ele permite atributos de valores múltiplos (como *nome_dependente* na Figura 6.25) e atributos compostos (como um atributo *endereço* com atributos componentes *rua* e *cidade*). Quando criamos tabelas de projetos E-R que contêm esses tipos de atributos, eliminamos essa subestrutura. Para atributos compostos, deixamos cada componente ser um atributo por si só. Para atributos de valores múltiplos, criamos uma tupla para cada item em um conjunto de valores múltiplos.

No modelo relacional, formalizamos essa idéia de que atributos não possuem qualquer subestrutura. Um domínio é **atômico** se os elementos do domínio são considerados unidades indivisíveis. Dizemos que um esquema de relação R está na **primeira forma normal** (1FN) se os domínios de todos os atributos de R são atômicos.

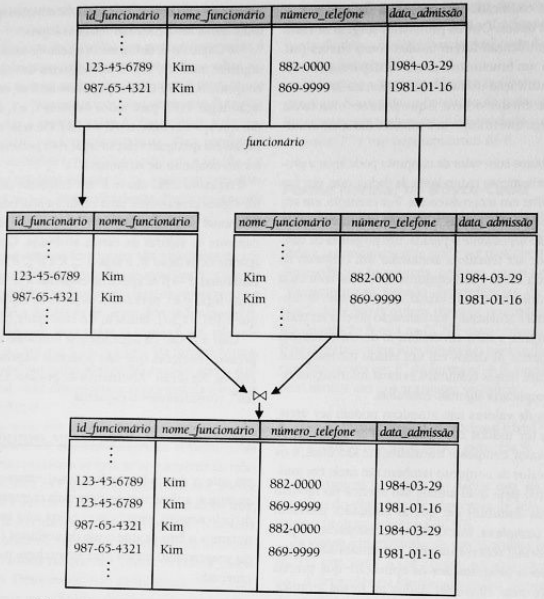


Figura 7.4 Perda de informações por meio de uma má decomposição.

Um conjunto de nomes é um exemplo de um valor não atômico. Por exemplo, se o esquema de uma relação *funcionário* incluísse um atributo *filhos* cujos elementos de domínio são conjuntos de nomes, o esquema não estaria na primeira forma normal.

Atributos compostos (como um atributo *endereço* com atributos componentes *rua* e *cidade*) também possuem domínios não atômicos.

Como os inteiros são considerados atômicos, o conjunto dos inteiros é um domínio atômico; o conjunto de todos os conjuntos de inteiros é um domínio não atômico. A diferença é que normalmente não consideramos que os inteiros têm subpartes, mas consideramos que conjuntos de inteiros têm subpartes – a saber, os inteiros que compõem o conjunto. Contudo, o ponto importante não é qual é o domínio em si, mas como usamos elementos do domínio em nosso banco de dados. O domínio de todos os inteiros seria não atômico se considerássemos cada inteiro como sendo uma lista ordenada de dígitos.

Como ilustração prática dessa questão, considere uma organização que atribui números de identificação de funcionários da seguinte forma: as primeiras duas letras especificam o departamento e os quatro dígitos restantes são um número único dentro do departamento do funcionário. Exemplos desses números seriam *CS0012* e *EE1127*. Esses números de identificação podem ser divididos em unidades menores e, portanto, são não atômicos. Se um esquema de relação tivesse um atributo cujo domínio consiste em números de identificação codificados como mostrado antes, o esquema não estaria na primeira forma normal.

Quando esses números de identificação são usados, o departamento de um funcionário pode ser encontrado escrevendo-se o código que divide a estrutura de um número de identificação. Fazer isso exige programação extra, e as

informações são codificadas no programa de aplicação e não no banco de dados. Outros problemas surgem se esses números de identificação forem usados como chaves primárias. Quando um funcionário muda de departamento, o número de identificação do funcionário precisa ser alterado em todo lugar em que ocorre, o que pode ser uma tarefa difícil, ou o código que interpreta o número daria um resultado incorreto.

O uso de atributos com valor de conjunto pode levar a projetos com armazenamento redundante de dados, que, por sua vez, podem resultar em inconsistências. Por exemplo, em vez do relacionamento entre contas e clientes ser representado como uma relação *depositante* separada, um projetista de banco de dados pode ser tentado a armazenar um conjunto de *titulares* com cada conta e um conjunto de *contas* com cada cliente. Sempre que uma conta é criada, ou o conjunto de titulares de uma conta é atualizado, a atualização precisa ser realizada em dois lugares; a falha em realizar as duas atualizações pode deixar o banco de dados em um estado inconsistente. Manter apenas um desses conjuntos evitaria informações repetidas, mas complicaria algumas consultas.

Alguns tipos de valores não atômicos podem ser úteis, embora devam ser usados com cuidado. Por exemplo, os atributos com valor composto normalmente são úteis, e os atributos com valor de conjunto também são úteis em muitos casos, motivo pelo qual ambos são aceitos no modelo E-R. Em muitos domínios em que as entidades possuem uma estrutura complexa, forçar uma representação de primeira forma normal representa uma responsabilidade desnecessária sobre o programador da aplicação, que precisa escrever código para converter dados na forma atômica. Também existe um overhead de tempo de execução em converter dados de e para a forma atômica. Portanto, o suporte para valores não atômicos pode ser muito útil nesses domínios. Na verdade, os sistemas de banco de dados modernos aceitam muitos tipos de valores não atômicos, como veremos no Capítulo 9. Entretanto, neste capítulo, nos limitamos às relações na primeira forma normal e, portanto, todos os domínios são atômicos.

Decomposição usando dependências funcionais

Na primeira seção deste capítulo, observamos que existe uma metodologia formal para avaliar se um esquema relacional deve ser decomposto. Essa metodologia é baseada nos conceitos das chaves e das dependências funcionais.

Chaves e dependências funcionais

As chaves e, mais geralmente, as dependências funcionais são restrições no banco de dados que exigem que relações satisfaçam certas propriedades. As relações que satisfazem todas essas restrições são relações **legais**.

No Capítulo 6, definimos a noção de uma *superchave* da seguinte maneira. Seja R um esquema de relação. Um subconjunto K de R é uma **superchave** de R se, em qualquer relação legal, $r(R)$, para todos os pares t_1 e t_2 de tuplas em r tais que $t_1 \neq t_2$, então, $t_1[K] \neq t_2[K]$. Ou seja, nenhum par de tuplas em qualquer relação legal $r(R)$ pode ter o mesmo valor no conjunto de atributos K.

Enquanto uma chave é um conjunto de atributos que identifica unicamente uma tupla inteira, uma dependência funcional permite expressar restrições que identificam unicamente os valores de certos atributos. Considere um esquema de relação R, e sejam $\alpha \subseteq R$ e $\beta \subseteq R$. A **dependência funcional** $\alpha \rightarrow \beta$ se aplica no esquema R se, em qualquer relação legal $r(R)$, para todos os pares de tuplas t_1 e t_2 em r tal que $t_1[\alpha] = t_2[\alpha]$, também é o caso que $t_1[\beta] = t_2[\beta]$.

Usar a noção de dependência funcional nos permite expressar restrições que não podemos expressar com superchaves. Na seção "Alternativa de projeto: Esquemas menores", consideramos o esquema

$$tom\_empr = (\underline{id\_cliente, número\_empréstimo}, quantia)$$

em que a dependência funcional *número_empréstimo* → *quantia* se aplica porque, para cada empréstimo (identificado pelo *número_empréstimo*) existe uma única quantia. Denotamos o fato de que o par de atributos (*id_cliente, quantia_empréstimo*) forma uma superchave para *tom_empr* escrevendo:

$$id\_cliente, número\_empréstimo \rightarrow id\_cliente, número\_empréstimo, quantia$$

ou, de modo equivalente,

$$id\_cliente, número\_empréstimo \rightarrow tom\_empr$$

Usaremos dependências funcionais de duas maneiras:

1. Para testar relações para ver se são legais sob um determinado conjunto de dependências funcionais. Se uma relação r é legal sob um conjunto F de dependências funcionais, dizemos que r **satisfaz** F.
2. Para especificar restrições no conjunto de relações legais. Portanto, nos concentraremos *apenas* nas relações que satisfazem um determinado conjunto de dependências funcionais. Se desejarmos nos restringir às relações no esquema R que satisfazem um conjunto F de dependências funcionais, dizemos que F **se aplica** em R.

Vamos considerar a relação r da Figura 7.5 para ver que dependências funcionais são satisfeitas. Observe que $A \rightarrow C$ é satisfeito. Existem duas tuplas que possuem um valor A de a_1. Essas tuplas possuem o mesmo valor C – a saber, c_1. De igual modo, as duas tuplas com um valor A de a_2 têm o mesmo valor C, c_2. Não existe qualquer outro par de tuplas distintas que têm o mesmo valor A. Entretanto, a dependência funcional $C \rightarrow A$ não é satisfeita. Para confirmar isso, considere as tuplas $t_1 = (a_2, b_3, c_2, d_3)$ e $t_2 = (a_3, b_3, c_2, d_4)$. Elas têm os mesmos valores C, c_2, mas possuem diferentes valores A, a_2 e a_3, respectivamente. Portanto, encontramos um par de tuplas t_1 e t_2 tal que $t_1[C] = t_2[C]$, mas $t_1[A] \neq t_2[A]$.

Algumas dependências funcionais são chamadas de **triviais** porque são satisfeitas por todas as relações. Por exemplo, $A \rightarrow A$ é satisfeito por todas as relações envolvendo o atributo A. Lendo a definição de dependência funcional literalmente, vemos que, para todas as tuplas t_1 e t_2 tais que $t_1[A] = t_2[A]$, é o caso que $t_1[A] = t_2[A]$. Da mesma forma, $AB \rightarrow A$ é satisfeito por todas as relações envolvendo o atributo A. Em geral, uma dependência funcional da forma $\alpha \rightarrow \beta$ é **trivial** se $\beta \subseteq \alpha$.

É importante notar que uma determinada relação pode, em qualquer momento, satisfazer algumas dependências funcionais que não precisam se aplicar no esquema da relação. Na relação *cliente* da Figura 2.4, vemos que *rua_cliente* → *cidade_cliente* é satisfeita. Entretanto, acreditamos que, no mundo real, duas cidades podem ter ruas com o mesmo nome. Assim, é possível, em algum momento, ter uma instância da relação *cliente* em que *rua_cliente* → *cidade_cliente* não é satisfeita. Desse modo, não incluiríamos *rua_cliente* → *cidade_cliente* no conjunto de dependências funcionais que se aplicam no esquema para a relação *cliente*.

Dado que um conjunto de dependências funcionais F se aplica em uma relação r, podemos inferir que algumas outras dependências funcionais também precisam se aplicar na relação. Por exemplo, dado um esquema $r = (A, B, C)$, se as dependências funcionais $A \rightarrow B$ e $B \rightarrow C$ se aplicam em r, podemos inferir que a dependência funcional $A \rightarrow C$ também precisa se aplicar em r. Pois, dado qualquer valor de A, pode haver apenas um valor correspondente para B, e para esse valor de B, pode haver apenas um valor correspondente para C. Mais adiante, na seção "Fechamento de um conjunto de dependências funcionais", estudaremos como fazer essas inferências.

Usaremos a notação F^+ para indicar o **fechamento** do conjunto F, ou seja, o conjunto de todas as dependências funcionais que podem ser inferidas dado o conjunto F. Claramente, F^+ é um superconjunto de F.

Forma normal de Boyce-Codd

Uma das formas normais mais desejáveis que podemos obter é a **Boyce-Codd Normal Form (BCNF)**. Ela elimina toda a redundância que pode ser descoberta com base nas dependências funcionais, embora, como veremos na seção "Decomposição usando dependências de valores múltiplos", pode haver outros tipos de redundância. Um esquema de relação R está na BCNF com respeito a um conjunto F de dependências funcionais se, para todas as dependências funcionais em F^+ da forma $\alpha \rightarrow \beta$, onde $\alpha \subseteq R$ e $\beta \subseteq R$, pelo menos um dos seguintes se aplica:

- $\alpha \rightarrow \beta$ é uma dependência funcional trivial (ou seja, $\beta \subseteq \alpha$).
- α é uma superchave para o esquema R.

Um projeto de banco de dados está na BCNF se cada membro do conjunto de esquemas de relação que constituem o projeto estiver na BCNF.

Já vimos um exemplo de um esquema que não está na BCNF, *tom_empr* = (*id_cliente, número_empréstimo, quantia*). A dependência funcional *número_empréstimo* → *quantia* se aplica em *tom_empr*, mas *número_empréstimo* não é uma superchave (pois, como você se lembra, um empréstimo pode ser feito a um consórcio de muitos clientes). Na seção "Alternativa de projeto: Esquemas menores", vimos que a decomposição de *tom_empr* em *tomador* e *empréstimo* foi um projeto melhor. O esquema *tomador* está na BCNF porque nenhuma dependência funcional trivial se aplica nele. O esquema *empréstimo* possui uma dependência funcional não trivial que se aplica, *número_empréstimo* → *quantia*, mas *número_empréstimo* é uma superchave (na

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Figura 7.5 Relação de exemplo r.

erdade, nesse caso, a chave primária) para *empréstimo*. ortanto, *empréstimo* está na BCNF.

Agora, exporemos uma regra geral para decompor esuemas que não estejam na BCNF. Seja R um esquema que ão está na BCNF. Portanto, existe pelo menos uma depenência funcional não trivial, $\alpha \rightarrow \beta$, tal que α não é uma superchave para R. Substituímos R em nosso projeto com dois squemas:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

No caso do *tom_empr* anterior, α = *número_empréstimo*, β = *quantia* e *tom_empr* é substituído por

- $(\alpha \cup \beta)$ = (*número_empréstimo*, *quantia*)
- $(R - (\beta - \alpha))$ = (*id_cliente*, *número_empréstimo*)

Nesse exemplo, ocorre que $\beta - \alpha = \beta$. Precisamos declarar a regra como fizemos para lidar corretamente com dependências funcionais que têm atributos que aparecem nos dois lados da seta. As razões técnicas para isso são discutidas mais adiante, na seção "Decomposição BCNF".

Quando decompomos um esquema que não está na BCNF, pode acontecer que um ou mais dos esquemas resultantes não estejam na BCNF. Nesses casos, mais decomposição é necessária, cujo resultado eventual é um conjunto de esquemas BCNF.

BCNF e a preservação de dependência

Vimos várias maneiras de expressar restrições de consistência de banco de dados: restrições de chave primária, dependências funcionais, restrições de verificação, afirmações e acionadores. Testar essas restrições cada vez que o banco de dados é atualizado pode ser uma tarefa custosa e, portanto, é útil projetar o banco de dados de uma forma que as restrições possam ser testadas eficientemente. Em especial, se o teste de uma dependência funcional puder ser feito considerando apenas uma relação, o custo de testar essa restrição é baixo. Veremos que a decomposição em BCNF pode impedir o teste eficiente de certas dependências funcionais.

Para ilustrar isso, suponha que façamos uma mudança aparentemente grande no modo como nossa instituição bancária opera. No projeto da Figura 6.25, um cliente pode ter apenas um funcionário como "banqueiro pessoal". Isso decorre do conjunto de relacionamento *banqueiro_cliente* ser muitos-para-um de *cliente* para *funcionário*, já que uma agência pode ter muitos funcionários, mas um funcionário pode trabalhar apenas em uma agência. A Figura 7.6 mostra um subconjunto da Figura 6.25, com essas adições.

Entretanto, existe uma falha nesse projeto. Ele permite que um cliente tenha dois (ou mais) banqueiros pessoais trabalhando para a mesma agência, algo que o banco não permite. Seria ideal se houvesse um único conjunto de relacionamento que pudéssemos referenciar para impor essa restrição. Isso exige que consideremos uma maneira diferente de mudar nosso projeto E-R. Em vez de incluirmos o conjunto de relacionamento *trabalha_em*, substituímos o conjunto de relacionamento *banqueiro_cliente* por um relacionamento ternário, *agência_banqueiro_cliente*, envolvendo os conjuntos de entidades *cliente*, *funcionário* e *agência*, que são muitos-para-um do par *cliente*, *funcionário* para *agência*, como mostra a Figura 7.7. Como esse projeto permite um único conjunto de relacionamento para representar a restrição, ele tem uma grande vantagem sobre o primeiro método que consideramos.

A comparação entre esses dois métodos, no entanto, não é tão clara. O esquema derivado de *agência_banqueiro_cliente* é

agência_banqueiro_cliente = (*id_cliente*, *id_funcionário*, *nome_agência*, *tipo*)

Como um funcionário pode trabalhar apenas em um banco, sabemos que na relação no esquema *agência_banqueiro_cliente* pode haver apenas um valor *nome_agência* associado a cada valor *id_funcionário*; ou seja:

$$id\_funcionário \rightarrow nome\_agência$$

Entretanto, somos obrigados a repetir o nome da agência para cada vez que um funcionário participa em um relacionamento *agência_banqueiro_cliente*. Vemos que *agência_banqueiro_cliente* não está na BCNF, pois *id_funcionário* não é uma superchave. Seguindo nossa regra para a decomposição BCNF, obtemos:

(*id_cliente*, *id_funcionário*, *tipo*)
(*id_funcionário*, *nome_agência*)

Esse projeto, que é exatamente igual ao nosso primeiro método usando o conjunto de relacionamento *trabalha_em*, dificulta impor a restrição de que um cliente pode ter no máximo um banqueiro pessoal em uma determinada agência. Podemos expressar essa restrição pela dependência funcional

$$id\_cliente, nome\_agência \rightarrow id\_funcionário$$

e observe que, em nosso projeto BCNF, não há um esquema que inclua todos os atributos aparecendo nessa dependência funcional. Como nosso projeto original torna computacionalmente difícil impor essa dependência funcional, dizemos que nosso projeto não é **preservador de depen-**

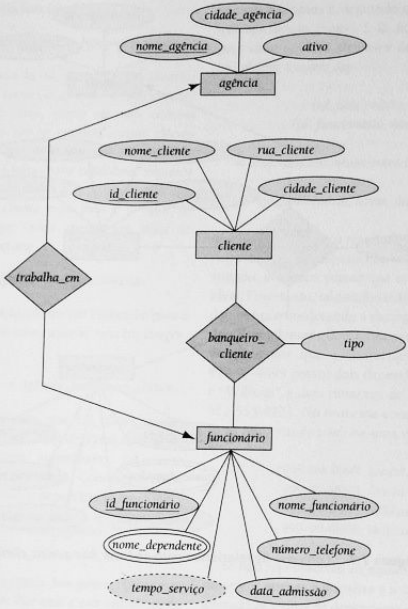


Figura 7.6 Conjuntos de relacionamento *cliente_banqueiro* e *trabalha_em*.

dência.[1] Como a preservação de dependência normalmente é considerada desejável, consideramos outra forma normal, mais fraca que a BCNF, que nos permitirá preservar dependências. Essa forma normal é chamada de terceira forma normal.[2]

Terceira forma normal

A BCNF exige que todas as dependências não triviais sejam da forma $\alpha \rightarrow \beta$, onde α é uma superchave. A terceira forma normal (3FN) atenua essa restrição permitindo dependências funcionais não triviais cujo lado esquerdo não é uma superchave. Antes de definirmos 3FN, lembremos que uma chave candidata é uma superchave mínima – ou seja, uma superchave no subconjunto apropriado do qual também é uma superchave.

Um esquema de relação R está na **terceira forma normal** no que se refere a um conjunto F de dependências funcionais se, para todas as dependências funcionais em F^+ da forma $\alpha \rightarrow \beta$, onde $\alpha \subseteq R$ e $\beta \subseteq R$, pelo menos uma das seguintes condições se aplica:

- $\alpha \rightarrow \beta$ é uma dependência funcional trivial.
- α é uma superchave para R.
- Cada atributo A em $\beta - \alpha$ está contido em uma chave candidata para R.

1. Tecnicamente, é possível que uma dependência cujos atributos não apareçam todos em esquema algum seja ainda imposta implicitamente, devido à presença de outras dependências que a implicam logicamente. Trataremos desse caso mais adiante, na seção "Preservação de dependência".
2. Você pode ter observado que pulamos a segunda forma normal. Ela tem significância apenas histórica e não é usada na prática.

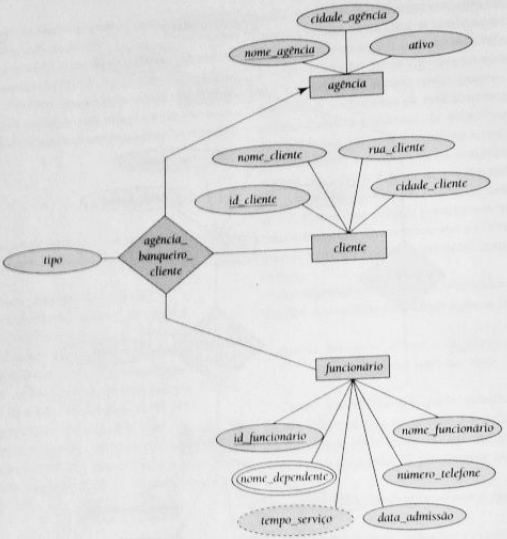


Figura 7.7 Conjunto de relacionamento *agência_banqueiro_cliente*.

Note que a terceira condição não afirma que uma única chave candidata deve conter todos os atributos em $\beta - \alpha$; cada atributo A em $\beta - \alpha$ pode estar contido em uma chave candidata *diferente*.

As primeiras duas alternativas são iguais às duas alternativas na definição de BCNF. A terceira alternativa da definição 3FN parece difícil de entender, e não é óbvio por que ela é útil. Ela representa, em certo sentido, um pequeno relaxamento das condições BCNF que ajuda a garantir que todo esquema tenha uma decomposição preservadora de dependência em 3FN. Sua finalidade se tornará mais clara posteriormente, quando estudarmos a decomposição em 3FN.

Observe que qualquer esquema que satisfaça a BCNF também satisfaz a 3FN, já que cada uma de suas dependências funcionais satisfaria uma das duas primeiras alternativas. A BCNF, portanto, é uma forma normal mais restritiva do que a 3FN.

A definição de 3FN permite certas dependências funcionais que não são permitidas na BCNF. Uma dependência $\alpha \rightarrow \beta$ que satisfaça apenas a terceira alternativa da definição 3FN não é permitida na BCNF, mas é permitida na 3FN.[3]

Agora, vamos considerar novamente *agência_banqueiro_cliente* e a dependência funcional

$$id\_funcionário \rightarrow nome\_agência$$

que fez com que o esquema não esteja na BCNF. Observe que α = *id_funcionário*, β = *nome_agência* e $\beta - \alpha$ = *nome_agência*. Ocorre que *nome_agência* está contido em uma chave candidata e, portanto, *agência_banqueiro_cliente* está na 3FN. Para mostrar isso, no entanto, é necessário um pouco de trabalho.

Sabemos que, além das dependências funcionais

$$id\_funcionário \rightarrow nome\_agência$$
$$id\_cliente, nome\_agência \rightarrow id\_funcionário$$

3. Essas dependências são exemplos de **dependências transitivas** (veja o Exercício prático 7.14). A definição original da 3FN era em termos de dependências transitivas. A definição que usamos é equivalente mas de compreensão mais fácil.

se aplicarem, a dependência funcional

id_cliente, id_funcionário → agência_banqueiro_cliente

se aplica como um resultado de (*id_cliente, id_funcionário*) ser a chave primária. Isso torna (*id_cliente, id_funcionário*) uma chave candidata. É claro, como ela não contém *nome_agência*, precisamos ver se existem outras chaves candidatas. Verificamos que o conjunto de atributos (*id_cliente, nome_agência*) é uma chave candidata. Vejamos por que esse é o caso.

Dado um valor de *id_cliente* específico e o valor de *nome_agência*, sabemos que existe apenas um valor de *id_funcionário* associado porque

id_cliente, nome_agência → id_funcionário

Então, para esse valor de *id_cliente* em especial e para o valor de *id_funcionário*, pode haver apenas uma tupla *agência_banqueiro* porque

id_cliente, id_funcionário → agência_banqueiro_cliente

Portanto, temos argumentado que (*id_cliente, nome_agência*) é uma superchave. Como nem *id_cliente* nem *nome_agência* isoladamente é uma superchave, (*id_cliente, nome_agência*) é uma chave candidata. Como essa chave candidata contém *nome_agência*, a dependência funcional

id_funcionário → nome_agência

não viola as regras da 3FN.

Nosso argumento de que *agência_banqueiro_cliente* está na 3FN exigiu algum esforço. Por esse e por outros motivos, é útil usar um método estruturado e formal para raciocinar sobre dependências funcionais, formas normais e decomposição de esquemas, o que fazemos na seção "Teoria da dependência funcional".

Vimos as trocas que precisam ser feitas entre BCNF e 3FN quando não há qualquer projeto BCNF de preservação de dependência. Essas trocas são descritas em mais detalhes na seção "Comparação entre BCNF e 3FN"; nessa seção, também descrevemos um método para verificação de dependência que permite obter as vantagens da BCNF e da 3FN.

Formas normais mais altas

Usar dependências funcionais para decompor esquemas pode não ser suficiente para evitar repetição desnecessária em certos casos. Considere uma ligeira variação na definição do conjunto de entidades *funcionário*, em que permitimos que os funcionários tenham vários números de telefone, alguns dos quais podem ser compartilhados por vários funcionários. Assim, *número_telefone* seria um atributo de valores múltiplos e, seguindo nossas regras para gerar esquemas de um projeto E-R, teríamos dois esquemas, um para cada um dos atributos de valores múltiplos, *número_telefone* e *nome_dep*:

(*id_funcionário, nome_dep*)
(*id_funcionário, número_telefone*)

Se fôssemos combinar esses esquemas para obter

(*id_funcionário, nome_dep, número_telefone*)

descobriríamos que o resultado está na BCNF, pois apenas dependências funcionais não triviais se aplicam. Como resultado, podemos pensar que essa combinação é uma boa idéia. Entretanto, tal combinação é uma má idéia, como podemos ver considerando o exemplo de um funcionário com dois dependentes e dois números de telefone. Por exemplo, vamos supor que o funcionário com o *id_funcionário* 999-99-9999 possui dois dependências chamados "David" e "William" e dois números de telefone, 512-555-1234 e 512-555-4321. No esquema combinado, precisamos repetir os números de telefone uma vez para cada dependente:

(999-99-9999, David, 512-555-1234)
(999-99-9999, David, 512-555-4321)
(999-99-9999, William, 512-555-1234)
(999-99-9999, William, 512-555-4321)

Se não repetíssemos os números de telefone e armazenássemos apenas a primeira e a última tupla, teríamos registrado os nomes de dependente e os números de telefone, mas as tuplas resultantes implicariam que David corresponde a 512-555-1234, enquanto William corresponde a 512-555-4321. Como sabemos, isso seria incorreto.

Como as formas normais baseadas em dependências funcionais não são suficientes para tratar situações como essa, foram definidas outras dependências e formas normais, que abordaremos nas seções "Decomposição usando dependências de valores múltiplos" e "Mais formas normais".

Teoria da dependência funcional

Vimos, em nossos exemplos, que é útil ser capaz de raciocinar sistematicamente sobre as dependências funcionais como parte de um processo de testar esquemas para BCNF ou 3FN.

Fechamento de um conjunto de dependências funcionais

Não é suficiente considerar o conjunto dado de dependências funcionais. Precisamos considerar *todas* as dependên-

cias funcionais que se aplicam. Veremos que, dado um conunto F de dependências funcionais, podemos provar que lgumas outras dependências funcionais se aplicam. Dizemos que essas dependências funcionais são "implicadas logicamente" por F.

Mais formalmente, dado um esquema relação R, uma dependência funcional f em R é **implicada logicamente** por um conjunto de dependências funcionais F em R se cada instância de relação $r(R)$ que satisfaz F também satisfaz f.

Suponha que temos um esquema de relação $R = (A, B, C, G, H, I)$ e o conjunto de dependências funcionais

$$A \rightarrow B$$
$$A \rightarrow C$$
$$CG \rightarrow H$$
$$CG \rightarrow I$$
$$B \rightarrow H$$

A dependência funcional

$$A \rightarrow H$$

é implicada logicamente. Ou seja, podemos mostrar que, sempre que nosso dado conjunto de dependências funcionais se aplicar em uma relação, $A \rightarrow H$ também precisa se aplicar na relação. Suponha que t_1 e t_2 sejam tuplas tais que

$$t_1[A] = t_2[A]$$

Como temos que $A \rightarrow B$, segue da definição de dependência funcional que

$$t_1[B] = t_2[B]$$

Então, como temos que $B \rightarrow H$, segue da definição de dependência funcional que

$$t_1[H] = t_2[H]$$

Portanto, mostramos que, sempre que t_1 e t_2 são tuplas tais que $t_1[A] = t_2[A]$, precisa ser que $t_1[H] = t_2[H]$. Todavia, essa é exatamente a definição de $A \rightarrow H$.

Seja F um conjunto de dependências funcionais. O **fechamento** de F, indicado por F^+, é o conjunto de todas as dependências funcionais implicadas logicamente por F. Dado F, podemos calcular F^+ diretamente da definição formal de dependência funcional. Se F fosse grande, esse processo seria longo e difícil. Esse cálculo de F^+ exige argumentos do tipo que acabamos de usar para mostrar que $A \rightarrow H$ está no fechamento de nosso conjunto de dependências de exemplo.

Os **axiomas**, ou regras de inferência, fornecem uma técnica mais simples de raciocinar sobre dependências funcionais. Nas regras que se seguem, usamos letras gregas ($\alpha, \beta, \gamma, \ldots$) para conjuntos de atributos e letras romanas maiúsculas do início do alfabeto para atributos individuais. Usamos $\alpha\beta$ para indicar $\alpha \cup \beta$.

Podemos usar as três regras seguintes para encontrar dependências funcionais implicadas logicamente. Aplicando essas regras *repetidamente*, podemos encontrar todos os F^+, dado F. Essa coleção de regras é chamada **axiomas de Armstrong** em homenagem à pessoa que a propôs originalmente.

- **Regra da reflexividade**. Se α é um conjunto de atributos e $\beta \subseteq \alpha$, então, $\alpha \rightarrow \beta$ se aplica.
- **Regra da expansão**. Se $\alpha \rightarrow \beta$ se aplica e γ é um conjunto de atributos, então, $\gamma\alpha \rightarrow \gamma\beta$ se aplica.
- **Regra da transitividade**. Se $\alpha \rightarrow \beta$ se aplica e $\beta \rightarrow \gamma$ se aplica, então, $\alpha \rightarrow \gamma$ se aplica.

Os axiomas de Armstrong são **confiáveis**, pois eles não geram quaisquer dependências funcionais incorretas. Eles são **completos**, pois, para um dado conjunto F de dependências funcionais, permitem gerar todo o F^+. As notas bibliográficas fornecem referências para provas de confiabilidade e completude.

Embora os axiomas de Armstrong sejam completos, é cansativo usá-los diretamente para o cálculo de F^+. Para simplificar mais as coisas, listamos regras adicionais. É possível usar os axiomas de Armstrong para provar que essas regras estão corretas (veja os Exercícios práticos 7.4 e 7.5 e o Exercício 7.21).

- **Regra da união**. Se $\alpha \rightarrow \beta$ se aplica e $\alpha \rightarrow \gamma$ se aplica, então, $\alpha \rightarrow \beta\gamma$ se aplica.
- **Regra da decomposição**. Se $\alpha \rightarrow \beta\gamma$ se aplica, então, $\alpha \rightarrow \beta$ se aplica e $\alpha \rightarrow \gamma$ se aplica.
- **Regra da pseudotransitividade**. Se $\alpha \rightarrow \beta$ se aplica e $\gamma\beta \rightarrow \delta$ se aplica, então, $\alpha\gamma \rightarrow \delta$ se aplica.

Vamos aplicar nossas regras ao exemplo do esquema $R = (A, B, C, G, H, I)$ e do conjunto F de dependências funcionais $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. Listamos vários números de F^+ aqui:

- $A \rightarrow H$. Como $A \rightarrow B$ e $B \rightarrow H$ se aplicam, aplicamos a regra da transitividade. Observe que foi muito mais fácil usar os axiomas de Armstrong para mostrar que $A \rightarrow H$ se aplica do que argumentar diretamente a partir das definições, como fizemos anteriormente nesta seção.
- $CG \rightarrow HI$. Como $CG \rightarrow H$ e $CG \rightarrow I$, a regra da união implica que $CG \rightarrow HI$.
- $AG \rightarrow I$. Como $A \rightarrow C$ e $CG \rightarrow I$, a regra da pseudotransitividade implica que $AG \rightarrow I$ se aplica.

```
F+ = F
repeat
  for each dependência funcional f em F+
    aplique as regras da refletividade e da expansão em f
    adicione a F+ as dependências funcionais resultantes
  for each par de dependências funcionais f1 e f2 em F+
    if f1 e f2 podem ser combinados usando transitividade,
      adicione a F+ a dependência funcional resultante
until F+ não alterar nada mais
```

Figura 7.8 Um procedimento para calcular F^+.

Outra maneira de descobrir que $AG \rightarrow I$ se aplica é a seguinte. Usamos a regra da expansão em $A \rightarrow C$ para inferir $AG \rightarrow CG$. Aplicando a regra da transitividade a essa dependência e $CG \rightarrow I$, inferimos $AG \rightarrow I$.

A Figura 7.8 mostra um procedimento que demonstra formalmente como usar os axiomas de Armstrong para calcular F^+. Nesse procedimento, quando uma dependência funcional é acrescentada a F^+, ela pode já estar presente, e, nesse caso, não há qualquer mudança para F^+. Na seção "Fechamento dos conjuntos de atributos", veremos uma maneira alternativa de calcular F^+.

Os lados esquerdo e direito de uma dependência funcional são subconjuntos de R. Como um conjunto de tamanho n possui 2^n subconjuntos, existe um total de $2 \times 2^n = 2^{n+1}$ dependências possíveis, onde n é o número de atributos em R. Cada repetição do loop repeat do procedimento, exceto a última repetição, acrescenta pelo menos uma dependência funcional a F^+. Portanto, o procedimento tem a garantia de terminar.

Fechamento dos conjuntos de atributos

Dizemos que um atributo B é **determinado funcionalmente** por α se $\alpha \rightarrow B$. Para testar se um conjunto α é uma superchave, precisamos criar um algoritmo para calcular o conjunto de atributos determinados funcionalmente por α. Uma maneira de fazer isso é calcular F^+, tomar todas as dependências funcionais com α como o lado esquerdo e tomar a união dos lados direitos de todas essas dependências. Entretanto, fazer isso pode ser oneroso, já que F^+ pode ser grande.

Um algoritmo eficiente para calcular o conjunto de atributos determinados funcionalmente por α é útil não só para testar se α é uma superchave, mas também para várias outras tarefas, como veremos mais tarde nesta seção.

Seja α um conjunto de atributos. Chamamos de "**fechamento de α sob F**" o conjunto de todos os atributos determinados funcionalmente por α sob um conjunto F de dependências funcionais; o indicamos por α^+. A Figura 7.9 mostra um algoritmo, escrito em pseudocódigo, para calcular α^+. A entrada é um conjunto F de dependências funcionais e o conjunto α de atributos. A saída é armazenada na variável *resultado*.

Para ilustrar como o algoritmo funciona, o usaremos para calcular $(AG)^+$ com as dependências funcionais definidas na seção anterior. Começamos com *resultado* = AG. A primeira vez que executamos o loop **while** para testar cada dependência funcional, descobrimos que

- $A \rightarrow B$ faz incluir B em *resultado*. Para ver esse fato, observamos que $A \rightarrow B$ está em F, $A \subseteq$ *resultado* (que é AG) e, logo, *resultado* := *resultado* $\cup$ B.
- $A \rightarrow C$ faz com que *resultado* se torne $ABCG$.
- $CG \rightarrow H$ faz com que *resultado* se torne $ABCGH$.
- $CG \rightarrow I$ faz com que *resultado* se torne $ABCGHI$.

Na segunda vez que executamos o loop **while**, nenhum novo atributo é acrescentado a *resultado* e o algoritmo termina.

```
resultado := α;
while (muda para resultado) do
  for each dependência funcional β → γ em F do
    begin
      if β ⊆ resultado then resultado := resultado ∪ γ;
    end
```

Figura 7.9 Um algoritmo para calcular α^+, o fechamento de α sob F.

Vejamos por que o algoritmo da Figura 7.9 está correto. primeira etapa está correta, já que $\alpha \rightarrow \alpha$ sempre se aplica ela regra da refletividade). Afirmamos que, para qualquer ibconjunto β de *resultado*, $\alpha \rightarrow \beta$. Como começamos o op **while** com $\alpha \rightarrow$ *resultado* sendo verdadeiro, podemos dicionar γ a *resultado* apenas se $\beta \subseteq$ *resultado* e $\beta \rightarrow \gamma$. ntão, *resultado* $\rightarrow \beta$ pela regra da refletividade e, portanto, $\rightarrow \beta$ pela transitividade. Outra aplicação da transitividade ostra que $\alpha \rightarrow \gamma$ (usando $\alpha \rightarrow \beta$ e $\beta \rightarrow \gamma$). A regra da união nplica que $\alpha \rightarrow$ *resultado* $\cup \gamma$ e, logo, α determina funcio- almente qualquer novo resultado gerado no loop **while**. ortanto, qualquer atributo retornado pelo algoritmo está m α^+.

É fácil ver que o algoritmo encontra todo α^+. Se houver ım atributo em α^+ que ainda não esteja em *resultado*, pre- isa haver uma dependência funcional $\beta \rightarrow \gamma$ para a qual β $\subseteq$ *resultado*, e pelo menos um atributo em γ não está em *esultado*.

Ocorre que, na pior hipótese, esse algoritmo pode tomar ım período de tempo quadrático no tamanho de F. Existe ım algoritmo mais rápido (embora um pouco mais com- plexo) que é executado linearmente no tempo no tamanho de F; esse algoritmo é apresentado como parte do Exercício prático 7.8.

Existem vários usos para o algoritmo de fechamento de atributo:

- Para testar se α é uma superchave, calculamos α^+ e veri- ficamos se α^+ contém todos os atributos de R.
- Podemos verificar se uma dependência funcional $\alpha \rightarrow \beta$ se aplica (ou seja, está em F^+), verificando se $\beta \subseteq \alpha^+$. Em outras palavras, calculamos α^+ usando o fechamento de atributo e, depois, verificamos se ele contém β. Esse tes- te é particularmente útil, como veremos mais adiante neste capítulo.
- Ele oferece uma maneira alternativa de calcular F^+: Para cada $\gamma \subseteq R$, encontramos o fechamento γ^+, e, para cada $S \subseteq \gamma^+$, geramos saída de uma dependência funcional $\gamma \rightarrow S$.

Cobertura canônica

Suponha que temos um conjunto de dependências funcio- nais F em um esquema de relação. Sempre que um usuário realiza uma atualização da relação, o sistema de banco de dados precisa garantir que a atualização não viole quais- quer dependências funcionais, ou seja, que todas as depen- dências funcionais em F sejam satisfeitas no novo estado do banco de dados.

O sistema precisa reverter a atualização se ela violar qualquer dependência funcional no conjunto F.

Podemos reduzir o esforço gasto em verificar violações testando um conjunto simplificado de dependências fun- cionais que tenha o mesmo fechamento do conjunto dado. Qualquer banco de dados que satisfaça o conjunto simplifi- cado de dependências funcionais também satisfará o con- junto original e vice-versa, já que os dois conjuntos pos- suem o mesmo fechamento. Entretanto, o conjunto simpli- ficado é mais fácil de testar. Veremos como o conjunto sim- plificado pode ser construído em breve. Primeiro, precisa- mos de algumas definições.

Dizemos que um atributo de uma dependência funcio- nal é **estranho** se pudermos removê-lo sem mudar o fecha- mento do conjunto de dependências funcionais. A defini- ção formal de **atributos estranhos** é a seguinte. Considere um conjunto F de dependências funcionais e a dependên- cia funcional $\alpha \rightarrow \beta$ em F.

- O atributo A é estranho em α se $A \in \alpha$, e F implica logi- camente $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- O atributo A é estranho em β se $A \in \beta$, e o conjunto de dependências funcionais $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ implica logicamente F.

Por exemplo, suponha que temos as dependências fun- cionais $AB \rightarrow C$ e $A \rightarrow C$ em F. Então, B é estranho em $AB \rightarrow C$. Como outro exemplo, suponha que temos as depen- dências funcionais $AB \rightarrow CD$ e $A \rightarrow C$ em F. Então, C seria estranho no lado direito de $AB \rightarrow CD$.

Cuidado com a direção das implicações quando usar a definição de atributos estranhos: se você mudar o lado es- querdo para o lado direito, a implicação *sempre* se aplicará. Ou seja, $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ sempre implicará logicamente F e, além disso, F sempre implicará logica- mente $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$.

Veja como podemos testar eficientemente se um atribu- to é estranho. Seja R o esquema de relação e seja F o conjun- to dado de dependências funcionais que se aplica em R. Considere um atributo A em uma dependência $\alpha \rightarrow \beta$.

- Se $A \in \beta$, para verificar se A é estranho, considere o con- junto $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ e verifique se $\alpha \rightarrow A$ pode ser inferido de F'. Para fazer isso, calcule α^+ (o fechamento de α) sob F'; se α^+ inclui A, então, A é estra- nho em β.
- Se $A \in \alpha$, para verificar se A é estranho, considere $\gamma = \alpha - \{A\}$ e verifique se $\gamma \rightarrow \beta$ pode ser inferido de F. Para isso, calcule γ^+ (o fechamento de γ) sob F; se γ^+ inclui todos os atributos em β, então, A é estranho em α.

Por exemplo, suponha que F contém $AB \rightarrow CD$, $A \rightarrow E$ e $E \rightarrow C$. Para verificar se C é estranho em $AB \rightarrow CD$, calcula- mos o fechamento de atributo de AB sob $F' = \{AB \rightarrow D, A \rightarrow E$ e $E \rightarrow C\}$. O fechamento é $ABCDE$, que inclui CD e, por- tanto, inferimos que C é estranho.

Uma **cobertura canônica** F_c para F é um conjunto de dependências tal que F implica logicamente todas as dependências em F_c e F_c implica logicamente todas as dependências em F. Além disso, F_c precisa ter as seguintes propriedades:

- Nenhuma dependência funcional em F_c contém um atributo estranho.
- Cada lado esquerdo de uma dependência funcional em F_c é único. Ou seja, não existem duas dependências $\alpha_1 \rightarrow \beta_1$ e $\alpha_2 \rightarrow \beta_2$ em F_c tais que $\alpha_1 = \alpha_2$.

Uma cobertura canônica para um conjunto de dependências funcionais F pode ser calculada como descrito na Figura 7.10. É importante notar que, quando estamos verificando se um atributo é estranho, a verificação usa as dependências no valor atual de F_c e não as dependências em F. Se uma dependência funcional contivesse apenas um atributo em seu lado direito, por exemplo $A \rightarrow C$, e esse atributo fosse considerado estranho, obteríamos uma dependência funcional com um lado direito vazio. Essas dependências funcionais devem ser excluídas.

A cobertura canônica de F, F_c, pode ser mostrada para ter o mesmo fechamento de F; portanto, testar se F_c é satisfeito é equivalente a testar se F é satisfeito. Entretanto, F_c é mínimo em certo sentido – ele não contém atributos estranhos e combina dependências funcionais com o mesmo lado esquerdo. É mais fácil testar F_c do que testar o próprio F.

Considere o seguinte conjunto F de dependências funcionais no esquema (A, B, C):

$$A \rightarrow BC$$
$$B \rightarrow C$$
$$A \rightarrow B$$
$$AB \rightarrow C$$

Vamos calcular a cobertura canônica para F.

- Existem duas dependências funcionais com o mesmo conjunto de atributos no lado esquerdo da seta:

$$A \rightarrow BC$$
$$A \rightarrow B$$

Combinamos essas dependências funcionais em $A \rightarrow BC$.

- A é estranho em $AB \rightarrow C$ porque F implica logicamente $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$. Essa afirmação é verdadeira porque $B \rightarrow C$ já está em nosso conjunto de dependências funcionais.
- C é estranho em $A \rightarrow BC$, já que $A \rightarrow BC$ é implicado logicamente por $A \rightarrow B$ e $B \rightarrow C$.

Assim, nossa cobertura canônica é

$$A \rightarrow B$$
$$B \rightarrow C$$

Dado um conjunto F de dependências funcionais, pode ocorrer que uma dependência funcional inteira no conjunto seja estranha, no sentido de que descartá-la não muda o fechamento de F. Podemos mostrar que uma cobertura canônica F_c de F não contém qualquer dependência funcional estranha. Suponha que, ao contrário, existe essa dependência funcional estranha em F_c. Os atributos do lado direito da dependência seriam, então, estranhos, o que não é possível pela definição de coberturas canônicas.

Uma cobertura canônica pode não ser única. Por exemplo, considere o conjunto de dependências funcionais $F = \{A \rightarrow BC, B \rightarrow AC$ e $C \rightarrow AB\}$. Se aplicarmos o teste da estranheza em $A \rightarrow BC$, descobriremos que tanto B quanto C são estranhos sob F. Entretanto, é incorreto excluir ambos! O algoritmo para encontrar a cobertura canônica seleciona um dos dois e o exclui. Então,

1. Se C for excluído, obtemos o conjunto $F' = \{A \rightarrow B, B \rightarrow AC$ e $C \rightarrow AB\}$. Agora, B não é estranho no lado direito de $A \rightarrow B$ sob F'. Continuando o algoritmo, descobrimos que A e B são estranhos no lado direito de $C \rightarrow AB$, levando a duas coberturas canônicas

$$F_c = \{A \rightarrow B, B \rightarrow C \text{ e } C \rightarrow A\} \text{ e}$$
$$F_c = \{A \rightarrow B, B \rightarrow AC \text{ e } C \rightarrow B\}$$

$F_c = F$
repeat
Use a regra da união para substituir quaisquer dependências em F_c da forma $\alpha_1 \rightarrow \beta_1$ e $\alpha_1 \rightarrow \beta_2$ por $\alpha_1 \rightarrow \beta_1\beta_2$.
Encontre uma dependência funcional $\alpha \rightarrow \beta$ em F_c com um atributo estranho em α ou em β.
/* Nota: O teste dos atributos estranhos é feito usando F_c, não F */
Se um atributo estranho for encontrado, exclua-o de $\alpha \rightarrow \beta$.
until F_c não mudar.

Figura 7.10 Calculando a cobertura canônica.

2. Se B for excluído, obtemos o conjunto $\{A \rightarrow C, B \rightarrow AC$ e $C \rightarrow AB\}$. Esse caso é simétrico ao caso anterior, levando às coberturas canônicas

$$F_c = \{A \rightarrow C, C \rightarrow B \text{ e } B \rightarrow A\} \text{ e}$$
$$F_c = \{A \rightarrow C, B \rightarrow C \text{ e } C \rightarrow AB\}$$

Como exercício, você pode encontrar mais uma cobertura canônica para F?

Decomposição sem perda

Seja R um esquema de relação e seja F um conjunto de dependências funcionais em R. Seja o par de relações R_1 e R_2 uma decomposição de R. Seja $r(R)$ uma relação com o esquema R. Dizemos que a decomposição é uma **decomposição sem perda** se, para todas as instâncias de banco de dados legais (ou seja, instâncias de banco de dados que satisfazem as dependências funcionais especificadas e outras restrições),

$$\Pi_{R1}(r) \bowtie \Pi_{R2}(r) = r$$

Em outras palavras, se projetarmos r para R_1 e R_2 e calcularmos a junção natural dos resultados da projeção, obtemos exatamente r. A decomposição contrária à sem perda é chamada de **decomposição com perda**. Algumas vezes, os termos **decomposição de junção sem perda** e **decomposição de junção com perda** são usados no lugar de decomposição sem perda e decomposição com perda.

Podemos usar dependências funcionais para mostrar quando certas decomposições são sem perda. Sejam R, R_1, R_2 e F como anteriormente. R_1 e R_2 formam uma decomposição sem perda de R se pelo menos uma das seguintes dependências funcionais está em F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

Em outras palavras, se $R_1 \cap R_2$ formar uma superchave de R_1 ou R_2, a decomposição de R será sem perda. Podemos usar o fechamento de atributo para testar eficientemente as superchaves, como vimos anteriormente.

Para ilustrar isso, considere o esquema

tom_empr = (*id_cliente, número_empréstimo, quantia*)

que decompusemos, na seção "Alternativa de projeto: Esquemas menores", em

tomador = (*id_cliente, número_empréstimo*)
empréstimo = (*número_empréstimo, quantia*)

Aqui, *tomador* ∩ *empréstimo* = *número_empréstimo* e *número_empréstimo* → *quantia*, satisfazendo a regra de decomposição sem perda.

Para o caso geral da decomposição de um esquema em múltiplos ao mesmo tempo, o teste para a decomposição sem perda é mais complicado. Veja as notas bibliográficas para obter referências sobre o assunto.

Embora o teste para a decomposição binária seja claramente uma condição suficiente para a decomposição sem perda, ela é uma condição necessária apenas se todas as restrições forem dependências funcionais. Veremos outros tipos de restrição mais adiante (em especial, um tipo de restrição chamado dependências de valores múltiplos, discutido na seção "Dependências de valores múltiplos"), que pode assegurar que uma decomposição será sem perda se nenhuma dependência funcional estiver presente.

Preservação de dependência

É mais fácil caracterizar preservação de dependência usando a teoria das dependências funcionais do que usando o método provisório que empregamos na seção "BCNF e a preservação de dependência".

Seja F um conjunto de dependências funcionais em um esquema R e seja $R_1, R_2, \ldots, R_n$ uma decomposição de R. A **restrição** de F para R_i é o conjunto F_i de todas as dependências funcionais em F^+ que incluem *apenas* atributos de R_i. Como todas as dependências funcionais em uma restrição envolvem atributos de apenas um esquema de relação, é possível testar essa dependência quanto à satisfação verificando apenas uma relação.

Observe que a definição da restrição usa todas as dependências em F^+, não apenas as que estão em F. Por exemplo, suponha $F = \{A \rightarrow B, B \rightarrow C\}$ e temos uma decomposição em AC e AB. A restrição de F para AC, então, é $A \rightarrow C$, já que $A \rightarrow C$ está em F^+, mesmo que não esteja em F.

O conjunto de restrições $F_1, F_2, \ldots, F_n$ é o conjunto das dependências que podem ser verificadas eficientemente. Agora, precisamos perguntar se testar apenas as restrições é o suficiente. Seja $F' = F_1 \cup F_2 \cup \ldots \cup F_n$. F' é um conjunto de dependências funcionais no esquema R, mas, em geral, $F' \neq F$. Entretanto, mesmo que $F' \neq F$, pode ocorrer que $F'^+ = F^+$. Se o último é verdadeiro, então, toda dependência em F é implicada logicamente por F' e, se verificarmos que F' é satisfeito, então verificamos que F é satisfeito. Dizemos que uma decomposição tendo a propriedade $F'^+ = F^+$ é uma **decomposição preservadora de dependência.**

A Figura 7.11 mostra um algoritmo para testar a preservação de dependência. A entrada é um conjunto $D = \{R_1, R_2, \ldots, R_n\}$ de esquemas de relação decompostos e um conjunto F de dependências funcionais. Esse algoritmo é one-

roso, já que exige o cálculo de F^+. Em vez de aplicar o algoritmo da Figura 7.11, consideramos duas alternativas.

Primeiro, observe que se cada membro de F puder ser testado em uma das relações da decomposição, então, a decomposição é preservadora de dependência. Essa é uma maneira fácil de mostrar preservação de dependência; entretanto, isso nem sempre funciona. Há casos em que, mesmo se a decomposição for preservadora de dependência, existe uma dependência em F que não pode ser testada em qualquer relação na decomposição. Portanto, esse teste alternativo pode ser usado apenas como uma condição suficiente que seja fácil de verificar; se ele falhar, não podemos concluir que a decomposição não é preservadora de dependência; em vez disso, precisaremos aplicar o teste geral.

Agora, forneceremos um segundo teste alternativo para a preservação de dependência que evita o cálculo de F^+. Explicaremos a idéia por trás do teste após apresentamos o teste em si. O teste aplica o seguinte procedimento a cada $\alpha \rightarrow \beta$ em F.

```
resultado = α
while (mudanças em resultado) do
  for each Ri na decomposição
    t = (resultado ∩ Ri)+ ∩ Ri
    resultado = resultado ∪ t
```

O fechamento de atributo aqui está sob o conjunto de dependências funcionais F. Se *resultado* contiver todos os atributos em β, então, a dependência funcional $\alpha \rightarrow \beta$ é preservada. A decomposição é preservadora de dependência se (e somente se) o procedimento mostrar que todas as dependências em F são preservadas.

As duas idéias básicas por trás desse teste são as seguintes.

- A primeira idéia é testar cada dependência funcional $\alpha \rightarrow \beta$ em F para ver se ela é preservada em F' (onde F' é como definido na Figura 7.11). Para fazer isso, calculamos o fechamento de α sob F'; a dependência é preservada exatamente quando o fechamento inclui β. A decomposição é preservadora de dependência se (e somente se) todas as dependências em F forem encontradas preservadas.
- A segunda idéia é usar uma forma modificada do algoritmo de fechamento de atributo para calcular o fechamento sob F' sem realmente primeiro calcular F'. Queremos evitar o cálculo de F', já que é bastante oneroso. Note que F' é a união de F_i, onde F_i é a restrição de F em R_i. O algoritmo calcula o fechamento de atributo de (*resultado* $\cap R_i$) com relação a F, intercepta o fechamento com R_i e adiciona o conjunto de atributos resultante a *resultado*; essa seqüência de etapas é equivalente a calcular o fechamento de *resultado* sob F_i. Repetir essa etapa para cada i dentro do loop while produz o fechamento de *resultado* sob F'.

Para entender por que esse método de fechamento de atributo modificado funciona corretamente, observamos que, para qualquer $\gamma \subseteq R_i$, $\gamma \rightarrow \gamma^+$ é uma dependência funcional em F^+, e $\gamma \rightarrow \gamma^+ \cap R_i$ é uma dependência funcional que está em F_i, a restrição de F^+ para R_i. De maneira oposta, se $\gamma \rightarrow \delta$ estivesse em F_i, então, δ seria um subconjunto de $\gamma^+ \cap R_i$.

Esse teste usa tempo polinomial em vez do tempo exponencial necessário para calcular F^+.

Decomposição usando dependências funcionais

Os esquemas de banco de dados reais são muito maiores do que os exemplos que cabem nas páginas de um livro. Por esse motivo, precisamos de algoritmos para a geração de projetos que estejam na forma normal apropriada. Nesta seção, apresentamos algoritmos para BCNF e 3FN.

```
calcule F+;
for each esquema Ri em D do
  begin
    Fi := a restrição de F+ para Ri;
  end
F' := ∅
for each restrição Fi do
  begin
    F' = F' ∪ Fi
  end
calcule F'+;
if (F'+ = F+) then retorne (true)
    else retorne (false);
```

Figura 7.11 Testando a preservação de dependência.

Decomposição BCNF

A definição de BCNF pode ser usada para testar diretamente se uma relação está na BCNF. Entretanto, o cálculo de F^+ pode ser uma tarefa tediosa. Primeiramente, descrevemos testes simplificados para verificar se uma relação está na BCNF. Se uma relação não estiver na BCNF, ela pode ser decomposta para criar relações que estejam na BCNF. Mais adiante nesta seção, descreveremos um algoritmo para criar uma decomposição sem perda de uma relação, de modo que a decomposição esteja na BCNF.

Testando a BCNF

Testar uma relação para ver se ela satisfaz a BCNF pode ser simplificado em alguns casos:

- Para verificar se uma dependência não trivial $\alpha \rightarrow \beta$ causa uma violação da BCNF, calcule α^+ (o fechamento de atributo de α) e verifique se ele inclui todos os atributos de R; ou seja, se ele é uma superchave de R.
- Para verificar se um esquema de relação R está na BCNF, basta verificar apenas as dependências no conjunto F dado quanto à violação da BCNF, em vez de verificar todas as dependências em F^+.

Podemos mostrar que, se nenhuma das dependências em F causar uma violação da BCNF, então, nenhuma das dependências em F^+ causará uma violação da BCNF.

Infelizmente, o último procedimento não funciona quando uma relação é decomposta. Ou seja, *não basta* usar F quando testar uma relação R_i em uma decomposição de R, quanto à violação da BCNF. Por exemplo, considere o esquema de relação R (A, B, C, D, E), com dependências funcionais F contendo $A \rightarrow B$ e $BC \rightarrow D$. Suponha que isso fosse decomposto em $R_1(A, B)$ e $R_2(A, C, D, E)$. Entretanto, como nenhuma das dependências em F contém apenas atributos de (A, C, D, E), podemos pensar erroneamente que R_2 satisfaz a BCNF. Na verdade, existe uma dependência $AC \rightarrow D$ em F^+ (que pode ser inferida usando a regra da pseudotransitividade das duas dependências em F), que mostra que R_2 não está na BCNF. Portanto, podemos precisar de uma dependência que esteja em F^+, mas não em F, para mostrar que uma relação decomposta não está na BCNF.

Um teste de BCNF alternativo algumas vezes é mais fácil do que calcular toda dependência em F^+. Para verificar se uma relação R_i em uma decomposição de R está na BCNF, aplicamos este teste:

- Para cada subconjunto α de atributos em R_i, verifique se α^+ (o fechamento de atributo de α sob F) não inclui qualquer atributo de $R_i - \alpha$, ou inclui todos os atributos de R_i.

Se a condição for violada por algum conjunto de atributos α em R_i, considere a seguinte dependência funcional, que pode se mostrar presente em F^+:

$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i.$$

Essa dependência mostra que R_i viola a BCNF.

Algoritmo de decomposição da BCNF

Agora, podemos apresentar um método geral para decompor um esquema de relação de modo a satisfazer a BCNF. A Figura 7.12 mostra um algoritmo para essa tarefa. Se R não estiver na BCNF, podemos decompor R em uma coleção de esquemas BCNF $R_1, R_2, \ldots, R_n$ pelo algoritmo. O algoritmo usa dependências que demonstram a violação da BCNF para realizar a decomposição.

A decomposição gerada pelo algoritmo não só está na BCNF como também é uma decomposição sem perda. Para

```
resultado := {R}
feito := false;
calcula F+;
while (not feito) do
    if (existe um esquema Ri em resultado que não esteja em BCNF)
        then begin
                Seja α → β uma dependência funcional não trivial que se aplica
                em Ri tal que α → Ri não esteja em F+ e α ∩ β = ∅;
                result := (result – Ri) ∪ (Ri – β) ∪ (α, β);
            end
        else done := true;
```

Figura 7.12 Algoritmo de decomposição da BCNF.

ver por que nosso algoritmo gera apenas declarações sem perda, observamos que, quando substituímos um esquema R_i por $(R_i - \beta)$ e (α, β), a dependência $\alpha \rightarrow \beta$ se aplica, e $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

Se não exigíssemos $\alpha \cap \beta = \varnothing$, então, os atributos em $\alpha \cap \beta$ não apareceriam no esquema $(R_i - \beta)$ e a dependência $\alpha \rightarrow \beta$ não mais se aplicaria.

É fácil ver que nossa decomposição de *tom_empr* na seção "Forma normal de Boyce-Codd" resultaria de aplicar o algoritmo. A dependência funcional *número_empréstimo* → *quantia* satisfaz a condição $\alpha \cap \beta = \varnothing$ e, portanto, seria escolhida para decompor o esquema.

O algoritmo de decomposição da BCNF usa tempo exponencial no tamanho do esquema inicial, já que o algoritmo para verificar se uma relação na decomposição satisfaz a BCNF pode usar tempo exponencial. As notas bibliográficas fornecem referências para um algoritmo que pode calcular uma decomposição de BCNF no tempo polinomial. Entretanto, o algoritmo pode "supernormalizar", ou seja, decompor uma relação desnecessariamente.

Como exemplo mais longo do uso do algoritmo de decomposição da BCNF, suponha que temos um projeto de banco de dados usando o seguinte esquema *fornecimento*:

fornecimento = (*nome_agência*, *cidade_agência*, *ativo*, *nome_cliente*, *número_empréstimo*, *quantia*)

O conjunto de dependências funcionais que exigimos que se aplique em *fornecimento* é

nome_agência → *ativo cidade_agência*
número_empréstimo → *quantia nome_agência*

Uma chave candidata para esse esquema é {*número_empréstimo*, *nome_cliente*}.

Podemos aplicar o algoritmo da Figura 7.12 no exemplo de *fornecimento* desta maneira:

- A dependência funcional

  nome_agência → *ativo cidade_agência*

  se aplica, mas *nome_agência* não é uma superchave. Portanto, *fornecimento* não está na BCNF. Substituímos *fornecimento* por

  agência = (*nome_agência*, *cidade_agência*, *ativo*)
  info_empréstimo = (*nome_agência*, *nome_cliente*, *número_empréstimo*, *quantia*)

- As únicas dependências funcionais não triviais que se aplicam em *agência* incluem *nome_agência* no lado esquerdo da seta. Como *nome_agência* é uma chave para *agência*, a relação *agência* está na BCNF.
- A dependência funcional

  número_empréstimo → quantia *nome_agência*

  se aplica em *info_empréstimo*, mas *número_empréstimo* não é uma chave para *info_empréstimo*. Substituímos *info_empréstimo* por

  empb = (*número_empréstimo*, *nome_agência*, *quantia*)
  tomador = (*nome_cliente*, *número_empréstimo*)

- *empb* e *tomador* estão na BCNF.

Portanto, a decomposição de *fornecimento* resulta nos três esquemas de relação *agência*, *empb* e *tomador*, cada um estando na BCNF. Você pode verificar que a decomposição é sem perda e preservadora de dependência.

Note que, embora o esquema *empb* anterior esteja na BCNF, poderíamos escolher decompô-lo mais usando a dependência funcional *número_empréstimo* → *quantia*, para obter os esquemas

empréstimo = (*número_empréstimo*, *quantia*)
agência_empréstimo = (*número_empréstimo*, *nome_agência*)

Esses correspondem aos esquemas que usamos neste capítulo.

Decomposição 3FN

A Figura 7.13 mostra um algoritmo para encontrar uma decomposição sem perda e preservadora de dependência para 3FN. O conjunto de dependências F_c usado no algoritmo é uma cobertura canônica para F. Observe que o algoritmo considera o conjunto dos esquemas R_j, $j = 1, 2, ..., i$; inicialmente, $i = 0$ e, nesse caso, o conjunto é vazio.

Vamos aplicar esse algoritmo em nosso exemplo da seção "BCNF e a preservação de dependência", em que mostramos que

agência_banqueiro_cliente = (*id_cliente*, *id_funcionário*, *nome_agência*, *tipo*)

está na 3FN mesmo que não esteja na BCNF. O algoritmo usaria as dependências funcionais em F, que, nesse caso, também é F_c:

(*id_cliente*, *id_funcionário* → *nome_agência*, *tipo*)
id_funcionário → *nome_agência*

e considere dois esquemas no loop **for**. A partir da primeira dependência funcional, o algoritmo gera como R_1 o esquema (*id_cliente*, *id_funcionário*, *nome_agência*, *tipo*). A partir

```
seja F_c uma cobertura canônica para F;
i := 0;
for each dependência funcional α → β em F_c do
  if nenhum dos esquemas R_j, j = 1, 2, ..., i contém α β
    then begin
      i := i + 1;
      R_i := α β;
    end
if nenhum dos esquemas R_j, j = 1, 2, ..., i contém uma chave candidata para R
  then begin
      i := i + 1;
      R_i := qualquer chave candidata para R;
  end
return (R_1, R_2, ..., R_i)
```

Figura 7.13 Decomposição sem perda e preservadora de dependência para 3FN.

da segunda, o algoritmo gera o esquema (*id_funcionário, nome_agência*) mas, então, não o cria como R_2 porque ele está contido em R_1 e, portanto, a condição if falha.

O algoritmo garante a preservação das dependências construindo explicitamente um esquema para cada dependência em uma cobertura canônica. Ele assegura que a decomposição é uma decomposição sem perda garantindo que pelo menos um esquema contenha uma chave candidata para o esquema sendo decomposto. O Exercício prático 7.12 fornece uma reflexão provando que isso basta para garantir uma decomposição sem perda.

Esse algoritmo também é chamado de **algoritmo de síntese 3FN**, já que ele toma um conjunto de dependências e adiciona um esquema de cada vez, em vez de decompor o esquema inicial repetidamente. O resultado não é definido de maneira única, pois um conjunto de dependências funcionais pode ter mais de uma cobertura canônica e, em alguns casos, o resultado do algoritmo depende da ordem em que ele considera a as dependências em F_c.

Se uma relação R_i está na decomposição gerada pelo algoritmo de síntese, então, R_i está na 3FN. Lembre-se de que, quando testamos a 3FN, basta considerar as dependências funcionais cujos lados direitos são um único atributo. Assim, para ver se R_i está na 3FN, você precisa se convencer de que qualquer dependência funcional $\gamma \rightarrow B$ que se aplica em R_i satisfaz a definição da 3FN. Considere que a dependência que gerou R_i no algoritmo de síntese é $\alpha \rightarrow \beta$. Agora, B precisa estar em α ou β, já que B está em R_i e $\alpha \rightarrow \beta$ gerou R_i. Vamos considerar os três casos possíveis:

- B está em α e β. Nesse caso, a dependência $\alpha \rightarrow \beta$ não teria estado em F_c, já que B seria estranho em β. Portanto, esse caso não se aplicaria.
- B está em β mas não em α. Considere dois casos:
  - γ é uma superchave. A segunda condição da 3FN é satisfeita.
  - γ não é uma superchave. Logo, α precisa conter algum atributo que não estejam em γ. Agora, como $\gamma \rightarrow B$ está em F^+, ele precisa ser derivável de F_c usando o algoritmo de fechamento de atributo em γ. A derivação não poderia ter usado $\alpha \rightarrow \beta$ (se ele tivesse usado, α precisaria estar contido no fechamento de atributo de γ, o que não é possível, já que consideramos que γ não é uma superchave). Agora, usando $\alpha \rightarrow (\beta - \{B\})$ e $\gamma \rightarrow B$, podemos derivar $\alpha \rightarrow \beta$ (já que $\gamma \subseteq \alpha\beta$, e γ não pode conter B porque $\gamma \rightarrow B$ não é trivial). Isso implicaria que B é estranho no lado direito de $\alpha \rightarrow \beta$, o que não é possível porque $\alpha \rightarrow \beta$ está na cobertura canônica F_c. Portanto, se B estiver em β, então, γ precisa ser uma superchave, e a segunda condição da 3FN precisa ser satisfeita.
- B está em α mas não em β.
  Como α é uma chave candidata, a terceira alternativa na definição da 3FN é satisfeita.

Curiosamente, o algoritmo que descrevemos para decomposição na 3FN pode ser implementado em tempo polinomial, ainda que testar uma dada relação para ver se ela satisfaz a 3FN seja não polinomial (o que significa que é muito improvável que um algoritmo de tempo polinomial seja inventado para essa tarefa).

Comparação entre BCNF e 3FN

Das duas formas normais para esquemas de banco de dados relacional, a saber, 3FN e BCNF, existem vantagens para a 3FN, na medida em que sabemos que sempre é possível ob-

ter um projeto 3FN sem sacrificar a preservação de dependência ou da ausência de perda. Contudo, existem desvantagens para a 3FN: Podemos precisar usar valores nulos para representar algumas das relações significativas possíveis entre itens de dados, e existe o problema da repetição de informações.

Nossos objetivos do projeto de banco de dados com dependências funcionais são:

1. BCNF
2. Ausência de perda
3. Preservação de dependência

Como nem sempre é possível satisfazer todos os três, podemos ser obrigados a escolher entre a BCNF e a preservação de dependência com a 3FN.

Vale observar que a SQL não oferece uma maneira de especificar dependências funcionais, exceto o caso especial de declarar superchaves usando as restrições **primary key** ou **unique**. É possível, embora um pouco complicado, escrever afirmações que impõem uma dependência funcional (veja o Exercício prático 7.9); infelizmente, testar as afirmações seria muito oneroso na maioria dos sistemas de banco de dados. Portanto, mesmo que tivéssemos uma decomposição preservadora de dependência, se usarmos a SQL padrão, podemos testar eficientemente apenas as dependências funcionais cujo lado direito é uma chave.

Embora o teste de dependências funcionais possa envolver uma junção se a decomposição não for preservadora de dependência, podemos reduzir o custo usando views materializadas, que muitos sistemas de banco de dados aceitam. Dada uma decomposição BCNF que não é preservadora de dependência, consideramos cada dependência em uma cobertura mínima F_c que não é preservada na decomposição. Para cada dependência $\alpha \rightarrow \beta$ desse tipo, definimos uma view materializada que calcula uma junção de todas as relações na decomposição e projeta o resultado em $\alpha\beta$. A dependência funcional pode ser facilmente testada na view materializada por meio de uma restrição **unique** (α). No lado negativo, existe um overhead de espaço e tempo devido à view materializada, mas, no lado positivo, o programador de aplicação não precisa se preocupar em escrever código para manter a consistência dos dados redundantes nas atualizações; é o trabalho do sistema de banco de dados manter a view materializada, ou seja, mantê-la atual quando o banco de dados é atualizado. (Mais adiante, na seção "Views materializadas" do Capítulo 14, descreveremos como um banco de dados pode realizar a manutenção de view materializada de modo eficiente.)

Desse modo, no caso de não sermos capazes de obter uma decomposição BCNF preservadora de dependência, normalmente é preferível optar pela BCNF e usar técnicas como views materializadas para reduzir o custo de verificar dependências funcionais.

Decomposição usando dependências de valores múltiplos

Alguns esquemas de relação, mesmo estando na BCNF, não parecem estar suficientemente normalizados, no sentido de que ainda experimentam o problema da repetição de informações. Considere novamente nosso exemplo de banco. Suponha que, em um projeto alternativo para o esquema de banco de dados bancário, tenhamos o esquema

cl_empr = (*número_empréstimo*, *id_cliente*, *nome_cliente*, *rua_cliente*, *cidade_cliente*)

O leitor astuto reconhecerá esse esquema como não BCNF, devido à dependência funcional

id_cliente → *nome_cliente*, *rua_cliente*, *cidade_cliente*

e porque *id_cliente* não é uma chave para *cl_empr*. Entretanto, considere que nosso banco está atraindo clientes de posses, com vários endereços (por exemplo, uma casa de inverno e uma casa de verão). Então, não desejamos mais impor a dependência funcional *id_cliente* → *rua_cliente cidade_cliente*, embora, é claro, ainda queiramos impor *id_cliente* → *nome_cliente* (ou seja, o banco não está lidando com clientes que operam sob vários nomes alternativos!) Seguindo o algoritmo de decomposição da BCNF, obtemos dois esquemas:

R_1 = (*id_cliente*, *nome_cliente*)
R_2 = (*número_empréstimo*, *id_cliente*, *rua_cliente*, *cidade_cliente*)

Ambos estão na BCNF (lembre-se de que os clientes não só podem ter mais de um empréstimo, mas também que um empréstimo pode ser feito a um grupo de pessoas e, portanto, nem *id_cliente* → *número_empréstimo* nem *número_empréstimo* → *id_cliente* se aplica).

Apesar de R_2 estar na BCNF, existe uma redundância. Repetimos o endereço de cada cliente uma vez para cada empréstimo que o cliente possui. Poderíamos resolver esse problema decompondo mais R_2 em:

empr_id_cl = (*número_empréstimo*, *id_cliente*)
resid_cl = (*id_cliente*, *rua_cliente*, *cidade_cliente*)

mas não há restrição que nos levaria a fazer isso.

Para lidar com esse problema, precisamos definir uma nova forma de restrição, chamada *dependência de valores múl-*

	α	β	$R = \alpha - \beta$
t_1	$a_1 \cdots a_i$	$a_{i+1} \cdots a_j$	$a_{j+1} \cdots a_n$
t_2	$a_1 \cdots a_i$	$b_{i+1} \cdots b_j$	$b_{j+1} \cdots b_n$
t_3	$a_1 \cdots a_i$	$a_{i+1} \cdots a_j$	$a_{j+1} \cdots a_n$
t_4	$a_1 \cdots a_i$	$b_{i+1} \cdots b_j$	$b_{j+1} \cdots b_n$

Figura 7.14 Representação tabular de $\alpha \twoheadrightarrow \beta$.

iplos. Como fizemos para as dependências funcionais, usare-nos as dependências de valores múltiplos para definir uma orma normal para esquemas de relação. Essa forma normal, hamada **quarta forma normal** (4FN), é mais restritiva do que a BCNF. Veremos que todo esquema 4FN também está na BCNF, mas nem todo esquema BCNF está na 4FN.

Dependências de valores múltiplos

As dependências funcionais excluem certas tuplas de uma relação. Se $A \rightarrow B$, então, não podemos ter duas tuplas com o mesmo valor A mas diferentes valores B. As dependências de valores múltiplos, por outro lado, não removem a existência de certas tuplas. Em vez disso, elas *exigem* que outras tuplas de uma determinada forma estejam presentes na relação. Por esse motivo, algumas vezes, as dependências funcionais são chamadas de **dependências de geração de igualdade** e as dependências de valores múltiplos são chamadas de **dependências de geração de tupla**.

Seja R um esquema de relação e seja $\alpha \subseteq R$ e $\beta \subseteq R$. A **dependência de valores múltiplos**

$$\alpha \twoheadrightarrow \beta$$

se aplica em R se, em qualquer relação legal $r(R)$, para todos os pares de tuplas t_1 e t_2 em r tais que $t_1[\alpha] = t_2[\alpha]$, existem tuplas t_3 e t_4 em r tais que

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

Essa definição é menos complicada do que parece. A Figura 7.14 fornece uma visão tabular de t_1, t_2, t_3 e t_4. Intuitivamente, a dependência de valores múltiplos $\alpha \twoheadrightarrow \beta$ diz que a relação entre α e β é independente da relação entre α e $R - \beta$. Se a dependência de valores múltiplos $\alpha \twoheadrightarrow \beta$ é satisfeita por todas as relações no esquema R, então, $\alpha \twoheadrightarrow \beta$ é uma dependência de valores múltiplos *trivial* no esquema R. Portanto, $\alpha \twoheadrightarrow \beta$ é trivial se $\beta \subseteq \alpha$ ou $\beta \cup \alpha = R$.

Para ilustrar a diferença entre dependências funcionais e de valores múltiplos, consideremos o esquema R_2 novamente e uma relação de exemplo nesse esquema mostrada na Figura 7.15. Precisamos repetir o número de empréstimo uma vez para cada empréstimo que um cliente possui. Essa repetição é desnecessária, uma vez que a relação entre um cliente e seu endereço é independente da relação entre esse cliente e um empréstimo. Se um cliente com id_cliente 99-123 possui um empréstimo (digamos, o de número L-23), queremos que esse empréstimo esteja associado a todos os endereços desse cliente. Portanto, a relação da Figura 7.16 é ilegal. Para torná-la legal, precisamos acrescentar as tuplas (L-23, 99-123, Main, Manchester) e (L-27, 99-123, North, Rye) à relação da Figura 7.16.

Comparando o exemplo anterior com nossa definição de dependência de valores múltiplos, vemos que desejamos que a dependência de valores múltiplos

$$id\_cliente \twoheadrightarrow rua\_cliente\ cidade\_cliente$$

se aplique. (A dependência de valores múltiplos *id_cliente* $\twoheadrightarrow$ *número_empréstimo* também se aplicará. Em breve veremos que elas são equivalentes.)

Como nas dependências funcionais, usaremos dependência de valores múltiplos de duas maneiras:

número_empréstimo	*id_cliente*	*rua_cliente*	*cidade_cliente*
L-23	99-123	North	Rye
L-23	99-123	Main	Manchester
L-93	15-106	Lake	Horseneck

Figura 7.15 Um exemplo de redundância em uma relação em um esquema BCNF.

número_empréstimo	*id_cliente*	*rua_cliente*	*cidade_cliente*
L-23	99-123	North	Rye
L-27	99-123	Main	Manchester

Figura 7.16 Uma relação R_2 ilegal.

1. Para testar relações para determinar se são legais sob um dado conjunto de dependências funcionais e de valores múltiplos
2. Para especificar restrições sobre as relações legais; portanto, nos concentraremos *apenas* nas relações que satisfazem um dado conjunto de dependências funcionais e de valores múltiplos

Observe que, se uma relação r não satisfaz uma determinada dependência de valores múltiplos, podemos construir uma relação r' que *satisfaça* a dependência de valores múltiplos acrescentando tuplas a r.

Seja D um conjunto de dependências funcionais e de valores múltiplos. O **fechamento** D^+ de D é o conjunto de todas as dependências funcionais e de valores múltiplos implicadas logicamente por D. Como fizemos para as dependências funcionais, podemos calcular D^+ a partir de D, usando as definições formais de dependências funcionais e dependência de valores múltiplos. Podemos seguir nesse raciocínio para dependência simples de valores múltiplos. Felizmente, as dependências de valores múltiplos que ocorrem na prática parecem muito simples. Para dependências complexas, é melhor raciocinar sobre conjuntos de dependências usando um sistema de regras de inferência. (A seção C.1.1 do Apêndice descreve um sistema de regras de inferência para dependências de valores múltiplos.)

Da definição de dependência de valores múltiplos, podemos propor a seguinte regra:

- Se $\alpha \rightarrow \beta$, então, $\alpha \twoheadrightarrow \beta$.

Em outras palavras, toda dependência funcional também é uma dependência de valores múltiplos.

Quarta forma normal

Considere novamente nosso exemplo do esquema BCNF

$$R_2 = (\underline{\textit{número\_empréstimo, id\_cliente}}, \textit{rua\_cliente}, \textit{cidade\_cliente})$$

em que a dependência de valores múltiplos *id_cliente* $\twoheadrightarrow$ *rua_cliente cidade_cliente* se aplica. Vimos nos parágrafos iniciais da seção "Decomposição usando dependências de valores múltiplos" que, embora esse esquema esteja na BCNF, o projeto não é ideal, já que precisamos repetir as informações de endereço de um cliente para cada empréstimo. Veremos que é possível usar a dependência de valores múltiplos dada para melhorar o projeto de banco de dados, decompondo esse esquema em uma decomposição de **quarta forma normal**.

Um esquema de relação R está na **quarta forma normal** (4FN) com respeito a um conjunto D de dependências funcionais e de valores múltiplos se, para todas as dependências de valores múltiplos em D^+ da forma $\alpha \twoheadrightarrow \beta$, onde $\alpha \subseteq R$ e $\beta \subseteq R$, pelo menos um dos seguintes se aplica

- $\alpha \twoheadrightarrow \beta$ é uma dependência de valores múltiplos trivial.
- α é uma superchave para o esquema R.

Um projeto de banco de dados está na 4FN se cada membro do conjunto de esquemas de relação que constitui o projeto estiver na 4FN.

Observe que a definição de 4FN difere da definição da BCNF apenas no uso das dependências de valores múltiplos em vez das dependências funcionais. Todo esquema 4FN está na BCNF. Para comprovar esse fato, notamos que, se um esquema R não está na BCNF, então, existe uma dependência funcional não trivial $\alpha \rightarrow \beta$ se aplicando em R, onde α não é uma superchave. Como $\alpha \rightarrow \beta$ implica $\alpha \twoheadrightarrow \beta$, R não pode estar na 4FN.

Seja R um esquema de relação e seja $R_1, R_2, \ldots, R_n$ uma decomposição de R. Para verificar se cada esquema de relação R_i na decomposição está na 4FN, precisamos descobrir quais dependências de valores múltiplos se aplicam em cada R_i. Lembre-se de que, para um conjunto F de dependências funcionais, a restrição F_i de F para R_i são todas as dependências funcionais em F^+ que incluem *apenas* atributos de R_i. Agora, considere um conjunto D de dependências funcionais e de valores múltiplos. A **restrição** de D para R_i é o conjunto D_i, consistindo de

1. Todas as dependências funcionais em D^+ que incluem apenas atributos de R_i
2. Todas as dependências de valores múltiplos da forma

$$\alpha \twoheadrightarrow \beta \cap R_i$$

onde $\alpha \subseteq R_i$ e $\alpha \twoheadrightarrow \beta$ está em D^+

```
resultado := {R};
feito := false;
calcula D+; Dado o esquema Ri, seja Di a restrição de D+ para Ri
while (not feito) do
  if (existe um esquema Ri em resultado que não esteja na 4FN em relação a Di)
    then begin
        seja α →→ β uma dependência de valores múltiplos não trivial que se aplica em Ri, tal
        que α → Ri não esteja em Di, e α ∩ β = ∅;
        resultado := (resultado – Ri) ∪ (Ri – β) ∪ (α, β);
end
    else feito := true;
```

Figura 7.17 Algoritmo de decomposição 4FN.

Decomposição 4FN

A analogia entre 4FN e BCNF se aplica ao algoritmo para decompor um esquema na 4FN. A Figura 7.17 mostra o algoritmo de decomposição 4FN. Ele é idêntico ao algoritmo de decomposição BCNF da Figura 7.12, exceto pelo fato de utilizar dependências de valores múltiplos em vez de dependências funcionais e usar a restrição de D^+ para R_i.

Se aplicarmos o algoritmo da Figura 7.17 a (*número_empréstimo, id_cliente, rua_cliente, cidade_cliente*), descobrimos que *id_cliente* →→ *número_empréstimo* é uma dependência de valores múltiplos não trivial e *id_cliente* não é uma superchave para o esquema. Seguindo o algoritmo, o substituímos por dois esquemas:

empr_id_cl = (*número_empréstimo, id_cliente*)
resid_cl = (*id_cliente, rua_cliente, cidade_cliente*)

Esse par de esquemas, que está na 4FN, elimina a redundância que encontramos anteriormente.

Como foi o caso quando estávamos lidando unicamente com dependências funcionais, estamos interessados nas declarações que são sem perda e que preservam dependências. O fato a seguir sobre dependências de valores múltiplos e ausência de perda mostra que o algoritmo da Figura 7.17 gera apenas declarações sem perda:

- Seja R um esquema de relação e seja D um conjunto de dependências funcionais e de valores múltiplos em R. Seja o par de relações R_1 e R_2 uma decomposição de R. Essa decomposição é sem perda de R se (e apenas se) pelo menos uma das seguintes dependências de valores múltiplos está em D^+:

$$R_1 \cap R_2 \rightarrow\rightarrow R_1$$
$$R_1 \cap R_2 \rightarrow\rightarrow R_2$$

Lembre-se de que afirmamos na seção "Decomposição sem perda" que, se $R_1 \cap R_2 \rightarrow R_1$ ou $R_1 \cap R_2 \rightarrow R_2$, então, R_1 e R_2 são uma decomposição sem perda de R. O fato anterior sobre dependências de valores múltiplos é uma instrução mais geral sobre ausência de perda. Ela diz que, para *cada* decomposição sem perda de R nos dois esquemas R_1 e R_2, uma das duas dependências $R_1 \cap R_2 \rightarrow\rightarrow R_1$ ou $R_1 \cap R_2 \rightarrow\rightarrow R_2$ precisa se aplicar.

O problema da preservação de dependência quando decompomos uma relação se torna mais complicado na presença de dependências de valores múltiplos. A seção C.1.2 do Apêndice aborda esse assunto.

Mais formas normais

A quarta forma normal não é de modo algum a forma normal "definitiva". Como vimos anteriormente, as dependências de valores múltiplos ajudam a entender e resolver algumas formas de repetição de informações que não podem ser entendidas em termos de dependências funcionais. Existem tipos de restrições, chamadas **dependências de junção**, que generalizam dependências de valores múltiplos e levam a outra forma normal chamada **forma normal de junção de projeção (FNJP)** (essa forma normal, em alguns livros, é chamada de **quinta forma normal**). Existe uma classe de restrições ainda mais gerais que leva a uma forma normal chamada **forma normal de chave de domínio (FNCD)**.

Um problema prático com o uso dessas restrições generalizadas é que a chave não só é difícil de analisar, como também não existe conjunto algum de regras de inferência completas para analisar as restrições. Portanto, a FNJP e a FNCD são usadas muito raramente. O Apêndice C fornece mais detalhes sobre essas formas normais.

Digna de nota pela sua ausência em nossa discussão de formas normais é a **segunda forma normal** (2FN). Não a abordamos porque seu interesse é unicamente histórico.

Simplesmente a definimos, e deixamos a experiência para você no Exercício prático 7.15.

Processo de projeto de banco de dados

Até agora, vimos problemas detalhados sobre as formas normais e a normalização. Nesta seção estudaremos como a normalização se encaixa no processo geral do projeto de banco de dados.

Anteriormente neste capítulo, iniciando na seção "Decomposição usando dependências funcionais", consideramos que um esquema de relação R é dado, e passamos a normalizá-lo. Há várias maneiras pelas quais poderíamos ter chegado ao esquema R:

1. R poderia ter sido gerado convertendo um diagrama E-R em um conjunto de esquemas de relação.
2. R poderia ter sido uma única relação contendo *todos* os atributos que são de interesse. O processo de normalização, então, desmembra R em relações menores.
3. R poderia ter sido o resultado de um projeto temporário de relações que, então, testamos para verificar se satisfaz uma forma normal desejada.

No restante desta seção, examinaremos as implicações desses métodos. Também veremos alguns problemas práticos no projeto de banco de dados, incluindo a desnormalização para desempenho e exemplos de maus projetos que não são detectados pela normalização.

Modelo E-R e normalização

Quando definimos cuidadosamente um diagrama E-R, identificando todas as entidades corretamente, os esquemas de relação gerados do diagrama E-R não precisam de muito mais normalização. Entretanto, pode haver dependências funcionais entre os atributos de uma entidade. Por exemplo, suponha que uma entidade *funcionário* tenha atributos *número_depto* e *endereço_depto* e que exista uma dependência funcional *número_depto* $\rightarrow$ *endereço_depto*. Poderíamos, então, normalizar a relação gerada de *funcionário*.

A maioria dos exemplos dessas dependências surge de um projeto ruim de diagrama E-R. Nesse exemplo, se tivéssemos projetado o diagrama E-R corretamente, teríamos criado uma entidade *depto* com o atributo *endereço_depto* e uma relação entre *funcionário* e *dpto*. Da mesma forma, uma relação envolvendo mais de duas entidades pode não estar em uma forma normal desejável. Como a maioria dos relacionamentos é binária, esses casos são relativamente raros. (Na verdade, algumas variantes do diagrama E-R realmente dificultam ou impossibilitam especificar relações não binárias.)

As dependências funcionais podem ajudar a detectar um mau projeto E-R. Se as relações geradas não estiverem na forma normal desejada, o problema pode ser corrigido no diagrama E-R. Ou seja, a normalização pode ser deixada por conta da intuição do projetista durante a modelagem E-R, e pode ser feita formalmente nas relações geradas do modelo E-R.

O leitor atento terá observado que, para ilustrarmos uma necessidade de dependências de valores múltiplos e a quarta forma normal, teríamos de começar com esquemas que não fossem derivados de nosso projeto E-R. De fato, o processo de criar um diagrama E-R costuma gerar projetos 4NF. Se uma dependência de valores múltiplos se aplica e não é implicada pela dependência funcional correspondente, ela geralmente surge das seguintes origens:

- Um relacionamento muitos-para-muitos
- Um atributo de valores múltiplos de um conjunto de entidade

Para um relacionamento muitos-para-muitos, cada conjunto de entidades relacionado possui seu próprio esquema, e existe um esquema adicional consistindo no atributo e na chave primária do conjunto de entidades (como no caso do atributo *nome_dependente* do conjunto de entidades *funcionário*).

O método de relação universal para projetos de banco de dados relacional começa com uma suposição de que existe um único esquema de relação contendo todos os atributos de interesse. Esse único esquema define como os usuários e as aplicações interagem com o banco de dados.

Nomeação dos atributos e relações

Um recurso desejável de um projeto de banco de dados é a **suposição de função única**, o que significa que cada nome de atributo possui um significado único no banco de dados. Isso evita que usemos o mesmo atributo para significar coisas diferentes em esquemas diferentes. Por exemplo, poderíamos considerar usar o atributo *número* para o número de empréstimo no esquema *empréstimo* e para o número de conta no esquema *conta*. A junção de uma relação no esquema *empréstimo* com uma em *conta* é significativa ("informações sobre pares empréstimo-conta onde o empréstimo e a conta calham de ter o mesmo número"). Embora os usuários e os desenvolvedores de aplicação possam trabalhar com cuidado para garantir o uso do *número* correto em cada circunstância, ter um nome de atributo diferente para o número de empréstimo e o número de conta serve para reduzir os erros do usuário. Realmente, observamos as suposições de função única em nossos projetos de banco de dados neste livro, e essa é uma boa prática geral a seguir.

Embora seja uma boa idéia manter nomes distintos para atributos incompatíveis, se os atributos de diferentes relações tiverem o mesmo significado, é possível usar o mesmo nome de atributo. Por exemplo, usamos os nomes de atributo *id_cliente* e *id_funcionário* nos conjuntos de entidades (e relações) *cliente* e *funcionário*. Se quiséssemos generalizar esses conjuntos de entidades criando um conjunto de entidades *pessoa*, precisaríamos renomear o atributo. Portanto, mesmo se não tivermos atualmente uma generalização de *cliente* e *funcionário*, se previmos essa possibilidade, é melhor usar o mesmo nome nos dois conjuntos de entidades (e relações).

A pesar de, tecnicamente, a ordem dos nomes de atributo em um esquema não importar, é uma convenção listar primeiro os atributos de chave primária. Isso torna a leitura da saída padrão (como de **select***) mais fácil.

Em grandes esquemas de banco de dados, os conjuntos de relacionamento (e os esquemas produzidos daí) normalmente são nomeados por uma concatenação dos nomes dos conjuntos de entidades relacionados, talvez com um hífen ou sublinhado. Usamos alguns desses nomes, por exemplo, *agência_conta* e *agência_empréstimo*. Usamos os nomes *tomador* e *depositante* em vez de nomes concatenados mais longos, como *empréstimo_cliente* ou *conta_cliente*. Isso foi aceitável porque não é difícil para você lembrar as entidades associadas de algumas relações. Nem sempre podemos criar nomes de relacionamento por uma simples concatenação; por exemplo, um relacionamento *gerente* ou *trabalha-para* entre funcionários não faria muito sentido se ele fosse denominado *funcionário_funcionário*! Da mesma forma, se houver vários conjuntos de relacionamento possíveis entre um par de conjuntos de entidades, os nomes de relacionamento precisam incluir pares extras para identificar o relacionamento.

Diferentes organizações possuem diferentes convenções para nomear entidades. Por exemplo, podemos chamar um conjunto de entidades de clientes de *cliente* ou *clientes*. Escolhemos usar a forma singular em nossos projetos de banco de dados. É aceitável usar o singular ou o plural desde que isso seja feito em todas as entidades de forma consistente.

À medida que os esquemas se tornarem maiores, com números de relacionamentos cada vez maiores, o uso de uma nomeação consistente de atributos, relacionamento e entidades facilitará muito a vida para o projetista de banco de dados e os programadores de aplicação.

Desnormalização para desempenho

Ocasionalmente, os projetistas de banco de dados escolhem um esquema que possui informações redundantes; ou seja, que não é normalizado. Eles usam a redundância para melhorar o desempenho de aplicações específicas. O ônus de não usar um esquema normalizado é o trabalho extra (em termos de tempo de codificação e execução) para manter os dados redundantes consistentes.

Por exemplo, suponha que o nome de um titular de conta precise ser exibido sempre com o número e o saldo da conta toda vez que a conta é acessada. Em nosso esquema normalizado, isso exige uma junção de *conta* com *depositante*.

Uma alternativa para calcular a junção instantânea é armazenar uma relação contendo todos os atributos de *conta* e *depositante*. Isso torna a exibição das informações da conta mais rápida. Entretanto, as informações de saldo para uma conta são repetidas para cada pessoa que possui a conta, e todas as cópias precisam ser atualizadas pela aplicação, sempre que o saldo da conta é atualizado. O processo de tomar um esquema normalizado e torná-lo não normalizado é chamado de **desnormalização**, e os projetistas o empregam a fim de ajustar o desempenho dos sistemas para aceitar operações em que o tempo é um fator crítico.

Uma alternativa melhor, aceita por muitos sistemas de banco de dados atuais, é usar o esquema normalizado e, adicionalmente, armazenar a junção de *conta* e *depositante* como uma view materializada. (Lembre-se de que uma view materializada é uma view cujo resultado é armazenado no banco de dados e atualizada quando as relações usadas na view são atualizadas.) Como a desnormalização, usar views materializadas envolve overheads de tempo e espaço; entretanto, esse método tem a vantagem de que manter a view atual é trabalho do sistema de banco de dados, não do programador da aplicação.

Outros aspectos de projeto

Existem algumas questões do projeto de banco de dados que não são resolvidas pela normalização e, portanto, podem levar a um mau projeto de banco de dados. Os dados relacionados ao tempo ou a intervalos de tempo possuem vários desses aspectos. Fornecemos alguns exemplos que, obviamente, devem ser evitados.

Considere um banco de dados de empresa, em que queremos armazenar o faturamento da empresa em diferentes anos. Uma relação *faturamento*(*id_empresa*, *ano*, *valor*) poderia ser usada para armazenar as informações de faturamento. A única dependência funcional nessa relação é *id_empresa*, *ano* → *valor*, e a relação está na BCNF.

Um projeto alternativo é usar várias relações, cada uma armazenando o faturamento de um ano diferente. Digamos que os anos de interesse sejam 2000, 2001 e 2002; teríamos, então, relações da forma *faturamento_2000*, *faturamento_2001*, *faturamento_2002*, todos estando no esquema (*id_empresa*, *faturamento*). A única dependência funcional aqui em cada relação seria *id_empresa* → *faturamento*; portanto, essas relações também estão na BCNF.

Todavia, esse projeto alternativo é claramente uma má idéia – precisaríamos criar uma nova relação todo ano e também teríamos de escrever novas consultas a cada ano para levar em consideração cada nova relação. As consultas também seriam mais complicadas, já que podem precisar se referir a muitas relações.

Ainda outra maneira de representar os mesmos dados é ter uma única relação *ano_empresa(id_empresa, faturamento_2000, faturamento_2001, faturamento_2002)*. Aqui, as únicas dependências funcionais são de *id_empresa* para os outros atributos, e novamente a relação está na BCNF. Esse projeto também é uma má idéia, uma vez que possui problemas semelhantes ao projeto anterior – ou seja, precisaríamos modificar o esquema de relação e escrever novas consultas todo ano. As consultas também seriam mais complicadas, já que podem precisar se referir a muitos atributos.

As representações como as da relação *ano_empresa*, com uma coluna para cada valor de um atributo, são chamadas **crosstabs**; elas são amplamente usadas em planilhas e relatórios e em ferramentas de análise de dados. Embora essas representações sejam úteis para exibir aos usuários, pelas razões já mencionadas, elas não são desejáveis em um projeto de banco de dados. Foram propostas extensões SQL de modo a converter dados de uma representação relacional normal para uma crosstab, para exibição.

Modelando dados temporais

Suponha que matemos dados em nosso banco mostrando não só o endereço de cada cliente, mas também todos os antigos endereços de que o banco tem ciência. Podemos, então, fazer consultas como "Encontre todos os clientes que moravam em Princeton em 1981". Nesse caso, podemos ter vários endereços para clientes. Cada endereço possui uma data inicial e uma data final associadas, indicando o período de tempo em que o cliente estava morando nesse endereço. Um valor especial para a data final, por exemplo, nulo, ou um valor em futuro distante, como 9999-12-31, podem ser usados para indicar que o cliente ainda está residindo no endereço.

Em geral, **dados temporais** são dados que possuem um intervalo de tempo associado, durante o qual eles são **válidos**.[4] Usamos o termo **instantâneo** dos dados para indicar o valor dos dados em um determinado ponto no tempo. Assim, um instantâneo dos dados de cliente fornece os valores de todos os atributos, como endereço, dos clientes em um dado momento no tempo.

Modelar dados temporais é uma tarefa complicada por várias razões. Por exemplo, suponha que tenhamos uma entidade *cliente* à qual desejamos associar um endereço variável no tempo. Para acrescentar informações temporais a um endereço, precisaríamos criar um atributo de valores múltiplos, cada um dos quais contendo um endereço e um intervalo de tempo. Além dos valores variáveis no tempo, as entidades podem, elas mesmas, ter um tempo válido associado. Por exemplo, uma entidade conta pode ter um tempo válido da data em que foi aberta até a data em que foi encerrada. Os relacionamentos também podem ter tempos válidos associados. Por exemplo, o relacionamento *depositante* entre um cliente e uma conta pode registrar quando o cliente se tornou o titular da conta. Então, poderíamos acrescentar intervalos de tempo válidos aos valores de atributo, entidades e relacionamentos. Incluir esse detalhe em um diagrama E-R o torna muito mais difícil criar e de compreender. Houve várias propostas de estender a notação E-R para especificar, de uma maneira simples, que um atributo ou relacionamento é variável no tempo, mas não existem quaisquer padrões aceitos.

Quando controlamos valores de dados pelo tempo, as dependências funcionais que consideramos se aplicar, como

$$id\_cliente \rightarrow rua\_cliente, cidade\_cliente$$

podem não mais se aplicar. A seguinte restrição (expressa em português) se aplicaria então: "Um *id_cliente* possui apenas um valor *rua_cliente* e *cidade_cliente* para um dado tempo t."

As dependências funcionais que se aplicam em um dado ponto no tempo são chamadas dependências funcionais temporais. Formalmente, uma **dependência funcional temporal** $X \xrightarrow{t} Y$ se aplica em um esquema de relação R se, para todas as instâncias legais r de R, todos os instantâneos de r satisfazem a dependência funcional $X \rightarrow Y$.

Poderíamos estender a teoria do projeto de banco de dados relacional para levar em consideração as dependências funcionais temporais. Entretanto, raciocinar com dependências funcionais regulares já é bastante difícil, e poucos projetistas estão preparados para lidar com dependências funcionais temporais.

Na prática, os projetistas de banco de dados se limitam aos métodos mais simples para projetar bancos de dados temporais. Uma técnica comumente usada é projetar o banco de dados inteiro (inclusive o projeto E-R e o projeto relacional) ignorando mudanças temporais (equivalentemente, levando apenas um instantâneo em consideração). Após isso, o projetista estuda as várias relações e decide quais delas exigem o acompanhamento da variação temporal.

4. Existem outros modelos de dados temporais que distinguem entre **tempo válido** e **tempo de transação**, este último registrando quando um fato foi registrado no banco de dados. Ignoramos esses detalhes para simplicidade.

A próxima etapa é incluir informações de tempo válido em cada uma dessas relações acrescentando tempo inicial e final como atributos. Por exemplo, considere que tenhamos uma relação

curso(id_curso, titulo_curso)

associando um título de curso a cada curso, que é identificado por um id_curso. O título do curso pode variar de acordo com o tempo, o que pode ser manipulado incluindo uma faixa de tempo válida; o esquema resultante seria

curso(id_curso, titulo_curso, inicio, fim)

Um exemplo dessa relação poderia ter dois registros (CS101, "Introdução à Programação", 1983-01-01, 2000-12-31) e (CS101, "Introdução a C", 2001-01-01, 9999-12-31). Se a relação for atualizada mudando o título do curso para "Introdução a Java", a data "9999-12-31" será atualizada para a data até que o valor antigo ("Introdução a C") seja válido, e uma nova tupla será acrescentada, contendo o novo título ("Introdução a Java"), com um tempo inicial apropriado.

Se outra relação tivesse uma chave estrangeira referenciando uma relação temporária, o projetista de banco de dados precisaria decidir se a referência é para a versão atual dos dados ou para os dados de um ponto específico no tempo. Por exemplo, uma relação que registra as atribuições de sala atuais para cada curso pode se referir implicitamente ao valor temporariamente atual associado a cada *id_curso*. Por outro lado, um registro no histórico de um aluno deve se referir ao título do curso à época em que o aluno fez o curso. Neste último caso, a relação referenciadora também precisa registrar informações de tempo, para identificar um determinado registro a partir da relação *curso*.

A chave primária original para uma relação temporal não mais identificaria unicamente uma tupla. Para resolver esse problema, poderíamos acrescentar os atributos de tempo inicial e final à chave primária. Entretanto, alguns problemas permaneceriam:

- É possível armazenar dados com intervalos sobrepostos, o que a restrição de chave primária não detectaria. Se o sistema aceitar um tipo de *tempo válido* nativo, ele poderá detectar e evitar essa sobreposição de intervalos de tempo.
- Para especificar uma chave estrangeira referenciando essa relação, as tuplas referenciadoras teriam de incluir os atributos de tempo inicial e final como parte de sua chave estrangeira, e os valores precisariam combinar com os da tupla referenciada. Além disso, se a tupla referenciada for atualizada (e o tempo final que estava no futuro for atualizado), a atualização precisa se propagar para todas as tuplas referenciadoras.

Se o sistema aceitar dados temporais de uma maneira melhor, poderemos permitir que a tupla referenciadora especifique um ponto no tempo, em vez de um período, e se baseie no sistema para garantir que haja uma tupla na relação referenciada cujo intervalo de tempo válido contém o ponto no tempo. Por exemplo, um registro de histórico pode especificar um *id_curso* e um tempo (digamos, a data inicial de um semestre), que é o bastante para identificar o registro correto na relação *curso*.

Como um caso especial comum, se todas as referências aos dados temporais são em relação apenas à data atual, uma solução mais simples é não incluir informações de tempo na relação, mas, em vez disso, criar uma relação *histórico* correspondente que tenha informações temporais, para valores do passado. Por exemplo, em nosso banco de dados de banco, poderíamos usar o projeto que criamos, ignorando mudanças temporais, para armazenar apenas as informações atuais. Uma informação histórica é movida para relações históricas. Portanto, a relação *cliente* pode armazenar apenas o endereço atual, enquanto uma relação *histórico_cliente* pode conter todos os atributos de *cliente*, com atributos *tempo_inicial* e *tempo_final* adicionais.

Embora não tenhamos oferecido qualquer método formal de lidar com dados temporais, os problemas que discutimos e os exemplos que fornecemos devem ajudá-lo a projetar um banco de dados que registre dados temporais. Outros problemas em manipular dados temporais, incluindo consultas temporais, são discutidos mais tarde, na seção 24.2.

Resumo

- Mostramos os riscos no projeto de banco de dados e como projetar sistematicamente um esquema de banco de dados que evite esses riscos. Os riscos incluem informações repetidas e a incapacidade de representar algumas informações.
- Mostramos o desenvolvimento de um projeto de banco de dados relacional a partir de um projeto E-R, quando o esquema pode ser combinado seguramente, e quando um esquema deve ser decomposto. Todas as decomposições válidas precisam ser sem perda.
- Descrevemos as suposições de domínios atômicos e a primeira forma normal.
- Apresentamos o conceito de dependências funcionais e o utilizamos para apresentar duas formas normais, a forma normal de Boyce-Cod (BCNF) e a terceira forma normal (3FN).
- Se a decomposição for preservadora de dependência, dado uma atualização de banco de dados, todas as de-

pendências funcionais podem ser verificáveis em relações individuais, sem calcular uma junção de relações da decomposição.

- Mostramos como raciocinar com dependências funcionais. Demos uma ênfase especial em quais dependências são implicadas logicamente por um conjunto de dependências. Também definimos a noção de uma cobertura canônica, que é um conjunto mínimo de dependências funcionais equivalente a um dado conjunto de dependências funcionais.
- Resumimos um algoritmo para decompor relações para a BCNF. Existem relações para as quais não há decomposição BCNF preservadora de dependência.
- Usamos as coberturas canônicas para decompor uma relação para a 3FN, que é um pequeno relaxamento da condição de BCNF. As relações na 3FN podem ter alguma redundância, mas há sempre uma decomposição preservadora de dependência para a 3FN.
- Apresentamos a noção de dependências de valores múltiplos, que especificam restrições que não podem ser especificadas apenas com dependências funcionais. Definimos a quarta forma normal (4FN) com dependências de valores múltiplos. A seção C.1.1 do Apêndice fornece detalhes do raciocínio sobre dependências de valores múltiplos.
- Outras formas normais, como FNJP e FNCD, eliminam formas de redundância mais sutis. Entretanto, essas são difíceis de lidar e raramente são usadas. O Apêndice C oferece detalhes sobre essas formas normais.
- Revisando as questões neste capítulo, note que a razão por que poderíamos definir métodos rigorosos para o projeto de banco de dados é que o modelo de dados relacional tem uma sólida base matemática. Essa é uma das principais vantagens do modelo relacional em comparação com outros modelos de dados que estudamos.

Termos de revisão

- Modelo E-R e normalização
- Decomposição
- Dependências funcionais
- Decomposição sem perda
- Primeira forma normal (1FN)
- Relações legais
- Superchave
- R satisfaz F
- F se aplica em R
- Forma normal de Boyce-Codd (BCNF)
- Preservação de dependência
- Terceira forma normal (3NF)
- Dependências funcionais triviais
- Fechamento de um conjunto de dependências funcionais
- Axiomas de Armstrong
- Fechamento de conjuntos de atributo
- Restrição de F para R_i
- Cobertura canônica
- Atributos estranhos
- Algoritmos de decomposição BCNF
- Algoritmo de decomposição 3FN
- Dependências de valores múltiplos
- Quarta forma normal (4FN)
- Restrição de uma dependência de valores múltiplos
- Forma normal de junção de projeção (FNJP)
- Forma normal de chave de domínio (FNCD)
- Relação universal
- Suposição de função única
- Desnormalização

Exercícios práticos

7.1 Suponha que decompomos o esquema $R = (A, B, C, D, E)$ em

$$(A, B, C)$$
$$(A, D, E)$$

Mostre que essa decomposição é uma decomposição sem perda se o seguinte conjunto F de dependências funcionais se aplicar:

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

7.2 Liste todas as dependências funcionais satisfeitas pela relação da Figura 7.18.

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Figura 7.18 Figura Relação do Exercício prático 7.2

7.3 Explique como as dependências funcionais podem ser usadas para indicar o seguinte:

- Um conjunto de relacionamento um-para-um existe entre os conjuntos de entidades *conta* e *cliente*.
- Um conjunto de relacionamento muitos-para-um existe entre os conjuntos de entidades *conta* e *cliente*.

7.4 Use os axiomas de Armstrong para provar a infalibilidade da regra da união. (*Dica*: use a regra da expansão para mostrar que, se $\alpha \rightarrow \beta$, então, $\alpha \rightarrow \alpha\beta$. Aplique a regra da expansão novamente, usando $\alpha \rightarrow \gamma$ e, depois, aplique a regra da transitividade.

7.5 Use os axiomas de Armstrong para provar a infalibilidade da regra da pseudotransitividade.

7.6 Calcule o fechamento do seguinte conjunto F de dependências funcionais para o esquema de relação $R = (A, B, C, D, E)$.

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

Liste as chaves candidatas para R.

7.7 Usando as dependências funcionais do Exercício prático 7.6, calcule a cobertura canônica F_c.

7.8 Considere o algoritmo da Figura 7.19 para calcular α^+. Mostre que esse algoritmo é mais eficiente do que o apresentado na Figura 7.9 (seção "Fechamento dos conjuntos de atributos") e que ele calcula α^+ corretamente.

7.9 Dado o esquema de banco de dados $R(a, b, c)$ e a relação r no esquema R, escreva uma consulta SQL para testar se a dependência funcional $b \rightarrow c$ se aplica na relação r. Escreva também uma afirmação SQL que imponha a dependência funcional. Considere que nenhum valor nulo está presente.

7.10 Seja $R_1, R_2, ..., R_n$ uma decomposição do esquema U. Seja $u(U)$ uma relação e seja $r_i = \Pi_{R_i}(u)$. Mostre que

$$u \subseteq r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

7.11 Mostre que a decomposição no Exercício prático 7.1 não é uma decomposição preservadora de dependência.

7.12 Mostre que é possível garantir que uma decomposição preservadora de dependência para 3FN é uma decomposição sem perda garantindo que pelo menos um esquema contém uma chave candidata para o esquema sendo decomposto. (*Dica*: Mostre que a junção de todas as projeções para os esquemas da decomposição não podem ter mais tuplas do que a relação original.)

7.13 Dê um exemplo de um esquema de relação R' e um conjunto F' de dependências funcionais tais que existam pelo menos três declarações sem perda distintas de R' para BCNF.

7.14 Seja um atributo *primo* que apareça em pelo menos uma chave candidata. Seja α e β conjuntos de atributos tais que $\alpha \rightarrow \beta$ se aplique, mas $\beta \rightarrow \alpha$ não se aplique. Seja A um atributo que não esteja em α, não esteja em β e para o qual $\beta \rightarrow A$ se aplique. Dizemos que A é *transitivamente dependente* de α. Podemos reformular nossa definição de 3FN desta maneira: um esquema de relação R está na 3FN com respeito a um conjunto F de dependências funcionais se não houver qualquer atributo não primo A em R para o qual A seja transitivamente dependente de uma chave para R.

Mostre que essa nova definição é equivalente à definição original.

7.15 Uma dependência funcional $\alpha \rightarrow \beta$ é chamada de **dependência parcial** se houver um subconjunto apropriado γ de α tal que $\gamma \rightarrow \beta$. Dizemos que β é *parcialmente dependente* de α. Um esquema de relação R está na **segunda forma normal** (2FN) se cada atributo A em R atender a um dos seguintes critérios:

- Aparece em uma chave candidata.
- Não é parcialmente dependente de uma chave candidata.

Mostre que todo esquema 3FN está na 2FN. (*Dica*: Mostre que toda dependência parcial é uma dependência transitiva.)

7.16 Dê um exemplo de um esquema de relação R e um conjunto de dependências tal que R esteja na BCNF mas não na 4FN.

Exercícios

7.17 Explique o que significa *repetição de informações* e *incapacidade de representar informações*. Explique por que cada uma dessas propriedades pode indicar um mau projeto de banco de dados relacional.

7.18 Por que certas dependências funcionais são chamadas de dependências funcionais *triviais*?

7.19 Use a definição de dependência funcional para argumentar que cada um dos axiomas de Armstrong (reflexividade, expansão e transitividade) é confiável.

7.20 Considere a seguinte regra proposta para dependências funcionais: se $\alpha \rightarrow \beta$ e $\gamma \rightarrow \beta$, então, $\alpha \rightarrow \gamma$. Prove que essa regra *não* é confiável mostrando uma relação r que satisfaz $\alpha \rightarrow \beta$ e $\gamma \rightarrow \beta$ mas não $\alpha \rightarrow \gamma$.

```
    resultado := ∅;
    /* fdcount é um array cujo iº elemento contém o número de atributos
       no lado esquerdo do iº FD que ainda não se sabe estarem em α⁺ */
    for i := 1 para |F| do
      begin
        seja β → γ o iº FD;
        fdcount [i] := |β|;
      end
    /* appears é um array com uma entrada para cada atributo. A entrada para o atributo A é uma lista de
       inteiros. Cada inteiro i na lista indica que A aparece no lado esquerdo do iº FD */
    for each atributo A do
      begin
        appears [A] := NIL;
        for i := 1 para |F| do
          begin
            seja β → γ o iº FD;
            if A ∈ β then acrescente i a appears [A];
          end
      end
    addin (α);
    return (resultado);

    procedure addin (α);
    for each atributo A em α do
      begin
        if A ∉ result then
          begin
            result := result ∪ {A};
            for each elemento i de appears [A] do
              begin
                fdcount [i] := fdcount [i] − 1;
                if fdcount [i] := 0 then
                  begin
                    seja β → γ o iº FD;
                    addin (γ);
                  end
              end
          end
      end
```

Figura 7.19 Um algoritmo para calcular α^+.

7.21 Use os axiomas de Armstrong para provar a confiabilidade da regra da decomposição.

7.22 Usando as dependências funcionais do Exercício prático 7.6, calcule B^+.

7.23 Mostre que a seguinte decomposição do esquema R do Exercício prático 7.1 não é uma decomposição sem perda:

(A, B, C)
(C, D, E)

Dica: dê um exemplo de uma relação r no esquema R tal que

$$\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$$

7.24 Cite os três objetivos de projeto para bancos de dados relacionais e explique por que cada um é desejável.

7.25 Forneça uma decomposição sem perda para BCNF do esquema R do Exercício prático 7.1.

7.26 Ao projetar um banco de dados relacional, por que podemos escolher um projeto não BCNF?

7.27 Forneça uma decomposição sem perda e preservadora de dependência para 3FN do esquema R do Exercício prático 7.1.

7.28 À luz dos três objetivos do projeto de banco de dados relacional, existe alguma razão para projetar um esquema de banco de dados que esteja na 2FN mas não esteja em nenhuma forma normal de ordem mais alta? (Veja no Exercício prático 7.15 a definição de 2FN.)

7.29 Dado um esquema de relação $r(A, B, C, D)$, $A \rightarrow\!\!\!\rightarrow BC$ implica logicamente $A \rightarrow\!\!\!\rightarrow B$ e $A \rightarrow\!\!\!\rightarrow C$? Se afirmativo, prove; caso contrário, forneça um contra-exemplo.

7.30 Explique por que a 4FN é uma forma normal mais desejável do que a BCNF.

Notas bibliográficas

A primeira abordagem da teoria do projeto de banco de dados relacional apareceu em um antigo documento de Codd [1970]. Nesse documento, Codd também apresentou as dependências funcionais e a primeira, segunda e terceira formas normais.

Os axiomas de Armstrong foram apresentados em Armstrong [1974]. Um desenvolvimento significante da teoria de banco de dados relacional ocorreu no final da década de 1970. Esses resultados estão reunidos em vários textos sobre a teoria de banco de dados, incluindo Maier [1983], Atzeni e Antonellis [1993] e Abiteboul e *et al.* [1995].

A BCNF foi introduzida em Codd [1972]. Biskup *et al.* [1979] fornecem o algoritmo que usamos para encontrar uma decomposição sem perda e preservadora de dependência para a 3FN. Resultados fundamentais sobre a propriedade de decomposição sem perda aparecem em Aho *et al.* [1979a].

Beeri *et al.* [1977] fornecem um conjunto de axiomas para dependências de valores múltiplos e provam que os axiomas dos autores são confiáveis e completos. As noções de 4FN, FNJP e FNCD são de Fagin [1977], Fagin [1979] e Fagin [1981], respectivamente. Veja as notas bibliográficas do Apêndice C para obter mais referências para literatura sobre normalização.

Jensen *et al.* [1994] apresentam um glossário de conceitos de banco de dados temporal. Gregersen e Jensen [1999] apresentam uma pesquisa das extensões à modelagem E-R para manipular dados temporais. Tansel *et al.* [1993] abordam a teoria, o projeto e a implementação de bancos de dados temporais. Jensen *et al.* [1996] descrevem extensões da teoria de dependência para dados temporais.

CAPÍTULO 8

Projeto de desenvolvimento de aplicação

Praticamente todo o uso dos bancos de dados ocorre de dentro dos programas de aplicação. De modo correspondente, quase toda a interação do usuário com bancos de dados é indireta, por meio de programas de aplicação. Portanto, não é surpresa que os sistemas de banco de dados há muito tempo tenham suporte para ferramentas como criadores de formulário e GUI, que ajudam no desenvolvimento rápido de aplicações que realizam interface com usuários. Nos últimos anos, a Web tem se tornado a interface com o usuário mais usada para bancos de dados.

Na primeira parte deste capítulo (seções "Interfaces de usuário e ferramentas" a "Servlets e JSP"), estudamos ferramentas e tecnologias que são necessárias para montar aplicações de banco de dados. Em particular, nos concentramos em ferramentas que ajudam no desenvolvimento de interfaces do usuário com bancos de dados. Começamos com uma visão geral das ferramentas para construir interfaces de formulário e relatórios. Depois, oferecemos uma visão geral detalhada de como desenvolver aplicações com interfaces baseadas na Web.

Mais adiante no capítulo, explicamos os triggers. Triggers permitem que as aplicações monitorem os eventos (atividades) de banco de dados e tomem ações quando ocorrem os eventos especificados. Eles também oferecem um meio de incluir regras e ações sem modificar o código de aplicação existente. Os triggers foram um acréscimo tardio ao padrão SQL. Apresentamos a sintaxe SQL conforme existe no padrão SQL:1999 e conforme existe em alguns sistemas comerciais.

Finalmente, abordamos a autorização e a segurança. Descrevemos os mecanismos de autorização fornecidos pela SQL e a sintaxe para o seu uso. Depois, discutimos as limitações dos mecanismos de autorização SQL e apresentamos outros conceitos e tecnologias exigidas para proteger bancos de dados e aplicações.

Interfaces de usuário e ferramentas

Embora a maioria das pessoas interaja com bancos de dados, poucas utilizam uma linguagem de consulta para interagir diretamente com um sistema de banco de dados. A maioria interage com um sistema de banco de dados por uma destas maneiras:

1. **Formulários e interfaces gráficas com o usuário** permitem que os usuários insiram valores que completam consultas predefinidas. O sistema executa as consultas, formata e exibe os resultados corretamente para o usuário. As interfaces gráficas com o usuário oferecem um modo fácil de interagir com o sistema de banco de dados.
2. **Geradores de relatório** permitem que relatórios predefinidos sejam gerados no conteúdo atual do banco de dados. Analistas ou gerentes vêem esses relatórios a fim de tomar decisões de negócios.
3. **Ferramentas de análise de dados** permitem que os usuários naveguem e analisem os dados interativamente.

Vale a pena observar que essas interfaces utilizam linguagens de consulta para se comunicarem com os sistemas de banco de dados.

Nesta seção, oferecemos uma visão geral dos formulários, interfaces gráficas com o usuário e geradores de relatório. O Capítulo 18 aborda as ferramentas de análise de dados com mais detalhes. Infelizmente, não existem pa-

drões para as interfaces com o usuário, e cada sistema de banco de dados normalmente oferece sua própria interface com o usuário. Nesta seção, descrevemos os conceitos básicos, sem entrar nos detalhes de qualquer produto específico de interface com o usuário.

Formulários e interfaces gráficas com o usuário

As interfaces de formulários são muito usadas para inserir dados nos bancos de dados e extrair informações dos bancos de dados, por meio de consultas predefinidas. Por exemplo, os mecanismos de busca da World Wide Web oferecem formulários que são usados para inserir palavras-chave. Pressionar um botão "submit" faz com que o mecanismo de busca execute uma consulta usando as palavras-chave inseridas e apresente o resultado ao usuário.

Como um exemplo mais orientado a banco de dados, você pode se conectar a um sistema de registro de universidade, no qual deverá preencher seu número de identificação e senha em um formulário. O sistema usa essa informação para verificar sua identidade, além de extrair informações do banco de dados, como seu nome e os cursos para os quais você se inscreveu, e exibi-las. Pode haver outros links na página Web que lhe permitam pesquisar cursos e localizar outras informações sobre os cursos, como os horários e o instrutor.

Os programadores podem criar formulários e interfaces gráficas com o usuário usando navegadores Web como front-end, ou usando formulários e outras facilidades fornecidas pelas interfaces de programação de aplicações (APIs), como Java Swing, ou APIs fornecidas com Visual Basic ou Visual C++. Navegadores Web que admitem HTML constituem os formulários e as interfaces gráficas com o usuário mais usados atualmente. Embora o navegador Web ofereça o front-end para interação do usuário, o processamento de back-end é feito no servidor Web, usando tecnologias como servlets Java, Java Server Pages (JSP) ou Active Server Page (ASP). Estudamos como criar formulários usando HTML na seção "HyperText Markup Language" e como criar sistemas de back-end usando servlets Java na seção "Servlets e JSP".

Há uma série de ferramentas que simplificam a criação de interfaces gráficas com o usuário e formulários, e permitem que os desenvolvedores de aplicação criem formulários em um padrão declarativo fácil, usando programas editores e formulários. Os usuários podem definir o tipo, o tamanho e o formato de cada campo em um formulário, usando o editor de formulários. As ações do sistema podem ser associadas às ações do usuário, como preencher um campo, pressionar uma tecla de função no teclado ou enviar um formulário. Por exemplo, a execução de uma consulta para preencher campos de nome e endereço pode ser associada ao preenchimento de um campo de número de identificação, e a execução de uma instrução de atualização pode estar associada à submissão de um formulário. Alguns exemplos desses sistemas são Oracle Forms, Sybase PowerBuilder e Oracle HTML-DB.

Verificações de erro simples podem ser realizadas definindo restrições sobre os campos no formulário. Por exemplo, uma restrição sobre um campo de data pode verificar se a data informada pelo usuário está formatada corretamente e se encontra em um intervalo desejado (por exemplo, um sistema de reserva pode exigir que a data não seja antes da data de hoje e não esteja a mais de seis meses no futuro). Embora essas restrições possam ser verificadas quando a transação for executada, a detecção de erros mais cedo ajuda o usuário a corrigir erros rapidamente. Os menus que indicam os valores válidos que podem ser inseridos em um campo podem ajudar a eliminar a possibilidade de muitos tipos de erros. Os desenvolvedores de sistemas descobrem que seu trabalho fica muito mais fácil, podendo controlar tais recursos de forma declarativa, com a ajuda de uma ferramenta de desenvolvimento de interface com o usuário, em vez de criar um formulário diretamente com o uso de uma linguagem de scripting ou programação.

Geradores de relatórios

Os geradores de relatórios são ferramentas que geram relatórios de resumo legíveis às pessoas a partir de um banco de dados. Eles integram a consulta do banco de dados com a criação de texto formatado e gráficos de resumo (como gráficos de barras ou pizza). Por exemplo, um relatório pode mostrar o total de vendas em cada um dos últimos dois meses para cada região de vendas.

O desenvolvedor de aplicações pode especificar formatos de relatório usando as facilidades de formatação do gerador de relatório. As variáveis podem ser usadas para armazenar parâmetros como o mês e o ano, além de definir campos no relatório. Tabelas, gráficos, gráficos de barra ou outros gráficos podem ser definidos por meio de consultas no banco de dados. As definições de consulta podem fazer uso dos valores de parâmetro armazenados nas variáveis.

Quando definimos uma estrutura de relatório em gerador de relatórios, podemos armazená-la e executá-la a qualquer momento para gerar um relatório. Os sistemas geradores de relatório oferecem uma série de facilidades para estruturar a saída tabular, como a definição de cabeçalhos de tabela e coluna, exibindo subtotais para cada grupo em uma tabela, dividindo automaticamente tabelas longas em várias páginas e exibindo subtotais ao final de cada página.

A Figura 8.1 é um exemplo de um relatório formatado. Os dados no relatório são gerados pela agregação em informações sobre pedidos.

Empresa de Suprimentos Acme Ltda.
Relatório Trimestral de Vendas

Período: 1º de janeiro a 31 de março de 2005

Região	Categoria	Vendas	Subtotal
Norte	Hardware de computador	1.000.000	
	Software de computador	500.000	
	Todas as categorias		1.500.000
Sul	Hardware de computador	200.000	
	Software de computador	400.000	
	Todas as categorias		600.000
		Total de vendas	2.100.000

Figura 8.1 Relatório formatado.

Há várias ferramentas de geração de relatórios oferecidas por vários fornecedores, como Crystal Reports e Microsoft (SQL Server Reporting Services). Vários pacotes de aplicações, como Microsoft Office, oferecem um meio de incorporar resultados de consulta formatados de um banco de dados diretamente em um documento. As facilidades de geração de gráficos oferecidas pela Crystal Reports, ou por planilhas como Excel, podem ser usadas para acessar dados de bancos de dados e gerar representações tabulares dos dados ou representações gráficas usando diagramas ou gráficos. Um ou mais desses gráficos podem ser incorporados dentro de documentos de texto usando, por exemplo, o Microsoft Word. Um recurso do Microsoft Office, chamado OLE (Object Linking and Embedding – vínculo e incorporação de objetos), é usado para vincular os gráficos ao documento de texto. O gráficos são criados inicialmente a partir dos dados gerados pela execução de consultas no banco de dados; as consultas podem ser reexecutadas e os gráficos gerados novamente quando forem necessários, para gerar uma versão atual do relatório geral.

Além de gerar relatórios estáticos, as ferramentas de geração de relatórios admitem a criação de relatórios interativos. Por exemplo, um usuário pode detalhar áreas de interesse, movendo de uma visão agregada, que mostra os totais de vendas para um ano inteiro, para os valores de vendas mensais em determinado ano. Retornaremos à análise interativa dos dados mais adiante, na seção "Análise de dados e OLAP" do Capítulo 18.

Interfaces Web para bancos de dados

A World Wide Web (Web, para abreviar) é um sistema de informações distribuídas, baseado em hipertexto. As interfaces Web para bancos de dados se tornaram muito importantes. Depois de esboçar vários motivos para realizar a interface dos bancos de dados com a Web, oferecemos uma visão geral da tecnologia Web (seção "Fundamentos da Web"). Depois, esboçamos técnicas para a criação de interfaces Web para bancos de dados, usando servlets e linguagens de scripting no servidor (seção "Servlets e JSP"). Fechamos esse assunto esboçando técnicas para a montagem de aplicações Web em grande escala e melhorando seu desempenho na seção "Montando grandes aplicações Web".

A Web tornou-se importante como um front-end para os bancos de dados por vários motivos: os navegadores Web oferecem um *front-end universal* para as informações fornecidas por back-ends localizados em qualquer lugar do mundo. O front-end pode executar em qualquer sistema de computador, e não existe necessidade de um usuário baixar qualquer software de uso especial a fim de acessar informações. Além do mais, hoje em dia, quase todos podem ter acesso a Web.

Com o crescimento dos serviços de informação e do comércio eletrônico na Web, os bancos de dados usados para serviços de informação, apoio à decisão e processamento de transação precisam estar ligados com a Web. A interface de formulários HTML é conveniente para o processamento de transações. O usuário pode preencher os detalhes em um formulário de pedido, depois clicar em um botão para enviar uma mensagem ao servidor. O servidor executa um programa de aplicação correspondente ao formulário de pedido, e essa ação, por sua vez, executa transações em um banco de dados no site do servidor. O servidor formata os resultados da transação e os envia de volta ao usuário.

Outro motivo para realizar a interface de bancos de dados com a Web é que a apresentação de documentos apenas

estáticos (fixos) em um site possui algumas limitações, mesmo quando o usuário não está fazendo qualquer consulta ou processamento de transação:

- Os documentos Web fixos não permitem que a exibição seja adaptada ao usuário. Por exemplo, um jornal pode querer adaptar sua apresentação com base em cada usuário, para dar destaque a artigos de notícias que provavelmente serão de interesse ao usuário.
- Quando os dados sobre uma organização são atualizados, os documentos Web baseados nos dados se tornam obsoletos se não forem atualizados simultaneamente. O problema se torna mais grave se vários documentos Web replicarem dados importantes, e todos precisarem ser atualizados.

Podemos consertar esses problemas gerando documentos Web dinamicamente a partir de um banco de dados. Quando um documento é solicitado, um programa é executado no site do servidor, que, por sua vez, executa consultas em um banco de dados e gera o documento solicitado com base nos resultados da consulta. Sempre que dados relevantes no banco de dados forem atualizados, os documentos gerados se tornarão automaticamente atualizados. O documento gerado também pode ser moldado ao usuário com base na informação do usuário armazenada no banco de dados.

As interfaces Web oferecem benefícios atraentes até mesmo para aplicações de banco de dados que são usadas apenas com uma única organização. O padrão da **HyperText Markup Language (HTML)** permite que o texto seja bem formatado, com informações importantes sendo destacadas. **Hyperlinks**, que são links para outros documentos, podem ser associados a regiões dos dados exibidos. O clique em um hyperlink busca e exibe o documento vinculado. Os hyperlinks são muito úteis para navegação de dados, permitindo que os usuários obtenham mais detalhes das partes dos dados conforme desejado.

Finalmente, os navegadores hoje podem apanhar programas junto com documentos HTML e executar os programas no navegador, de modo seguro – ou seja, sem danificar dados no computador do usuário. Os programas podem ser escritos em linguagens de scripting no servidor, como JavaScript, ou podem ser "applets" escritos na linguagem Java, ou animações escritas em linguagens como Flash ou Shockwave. Esses programas permitem a construção de sofisticadas interfaces com o usuário, além daquilo que é possível apenas com HTML; interfaces que podem ser usadas sem o download e a instalação de qualquer software. Assim, as interfaces Web são poderosas e atraentes visualmente, e encobriram interfaces com o usuário de uso especial para uma série de aplicações de banco de dados.

Fundamentos da Web

Aqui, revisamos parte da tecnologia fundamental por trás da World Wide Web, para leitores que não estão familiarizados com essa tecnologia.

Uniform Resource Locators

Um **localizador** de recurso uniforme (URL – Uniform Resource Locator) é um nome globalmente exclusivo para cada documento que pode ser acessado na Web. Um exemplo de um URL é

```
http://www.acm.org/sigmod
```

A primeira parte do URL indica como o documento deve ser acessado: "http" significa que o documento deve ser acessado pelo HyperText Transfer Protocol (protocolo de transferência de hipertexto), que é um protocolo para a transferência de documentos HTML. A segunda parte mostra o nome de uma máquina que possui um servidor Web. O restante do URL é o nome de caminho do arquivo na máquina, ou outro identificador exclusivo de um documento dentro da máquina.

Muitos dados na Web são gerados dinamicamente. Um URL pode conter o identificador de um programa localizado na máquina servidora da Web, além de argumentos a serem dados ao programa. Um exemplo de tal URL é

```
http://www.google.com/search?q=silberschatz
```

que diz que o programa `search` no servidor `www.google.com` deve ser executado com o argumento `q=silberschatz`. O programa é executado, usando os argumentos dados, e retorna um documento HTML, que é enviado para o front-end.

HyperText Markup Language

A Figura 8.2 é um exemplo do código-fonte de um documento HTML. A Figura 8.3 mostra a imagem exibida que esse documento cria.

As figuras mostram como a HTML pode exibir uma tabela e um formulário simples, para que os usuários selecionem o tipo (conta ou empréstimo) a partir de um menu, e informem um número em uma caixa de texto. A HTML também admite vários outros tipos de entrada. Um clique no botão submit faz com que o programa `BankQuery` (especificado no campo form `action`) seja executado no servidor Web, com os valores fornecidos pelo usuário para os argumentos `type` e `number` (especificado nos campos `select` e `input`). O programa gera um documento HTML, que é então enviado de volta e exibido ao usuário; veremos como construir esses programas nas seções 8.3.4, "Servlets e JSP" e 8.4.5.

```
<html>
<body>
<table BORDER COLS=3>
<tr> <td>A-101</td> <td>Downtown  </td> <td>500</td> </tr>
<tr> <td>A-102</td> <td>Perryridge</td> <td>400</td> </tr>
<tr> <td>A-201</td> <td>Brighton  </td> <td>900</td> </tr>
</table>
<center> A relação <i>conta</i></center>

<form action="BankQuery" method=get>
Selecione conta/empréstimo e informe número <br>>
<select name="type">
    <option value="conta" selected>Conta </option>
    <option value="empréstimo">            Empréstimo </option>
</select>
<input type=text size=5 name="number">
<input type=submit value="submit">
</form>
</body>
</html>
```

Figura 8.2 Um texto fonte HTML.

O HTTP define duas maneiras como os valores informados por um usuário no navegador podem ser enviados ao servidor Web. O método `get` codifica os valores como parte do URL. Por exemplo, se a página de busca do Google usasse um formulário com um parâmetro de entrada chamado `q` com o método `get`, e o usuário digitasse a string "silberschatz", submetendo o formulário, o navegador solicitaria o seguinte URL do servidor Web:

```
http://www.google.com/search?q=silberschatz
```

O método `post`, em vez disso, enviaria uma solicitação para a página `www.google.com` e enviaria os valores de parâmetro como parte da troca do protocolo HTTP entre o servidor Web e o navegador. O formulário na Figura 8.2 especifica que o formulário usa o método `get`.

Embora o código HTML possa ser criado usando um editor de textos simples, existem diversos editores que permitem a criação de texto HTML por meio de uma interface gráfica. Esses editores permitem que construções como formulários, menus e tabelas sejam inseridas no documento HTML a partir de um menu de opções, em vez de ter de digitar o código para gerar as construções.

A HTML aceita *folhas de estilo*, que podem alterar as definições padrão de como uma construção de formatação HTML é exibida, além de outros atributos de exibição como cor de segundo plano da página. O padrão de *folha de estilo em cascata* (CSS – Cascading Stylesheet) permite que a mesma folha de estilo seja usada para vários documentos HTML, dando uma aparência distinta, porém uniforme, a todas as páginas em um site.

Scripting no cliente e applets

A incorporação de código de programa nos documentos permite que as páginas Web sejam **ativas**, executando atividades como animação pela execução de programas no site local, em vez de simplesmente apresentando texto e gráfi-

A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900

A relação conta

Selecione conta/empréstimo e informe número

Conta | submit

Figura 8.3 Exibição do fonte HTML da Figura 8.2.

cos passivos. O uso principal de tais programas é a interação flexível com o usuário, além do poder de interação limitado fornecido pela HTML e por formulários HTML. Além disso, a execução de programas no cliente agiliza bastante a interação, em comparação com cada interação sendo enviada para ser processada em um site servidor.

Um perigo no suporte a tais programas é que, se o projeto do sistema for feito sem muito cuidado, o código de programa embutido em uma página Web (ou, de forma equivalente, em uma mensagem de correio eletrônico) pode realizar ações maliciosas no computador do usuário. As ações maliciosas podem variar desde a leitura de informações privadas e a exclusão ou modificação de informações no computador, até tomar o controle do computador e propagar o código para outros computadores (através de e-mail, por exemplo). Diversos vírus de e-mail se espalharam muito nos últimos anos dessa maneira.

Um dos motivos para a linguagem *Java* se tornar tão popular é que ela oferece um modo seguro para execução de programas nos computadores dos usuários. O código Java pode ser compilado em um "código de bytes" independente de plataforma, que pode ser executado em qualquer navegador que admite Java. Diferente dos programas locais, os programas Java (applets) baixados como parte de uma página Web não têm autoridade para realizar quaisquer ações que poderiam ser destrutivas. Eles têm permissão para exibir dados na tela, ou fazer uma conexão de rede com o servidor do qual a página foi baixada, a fim de obter mais informações. Porém, eles não têm permissão para acessar arquivos locais, executar quaisquer programas do sistema ou fazer conexões de rede com quaisquer outros computadores.

Embora Java seja uma linguagem de programação completa, existem linguagens mais simples, chamadas **linguagens de scripting**, que podem enriquecer a interação com o usuário, oferecendo a mesma proteção da Java. Essas linguagens oferecem construções que podem ser embutidas com um documento HTML. As **linguagens de scripting no cliente** são criadas para serem executadas no navegador Web do cliente. Dessas, a linguagem *JavaScript* é de longe a mais utilizada. JavaScript normalmente é utilizada para diversas tarefas. Por exemplo, funções escritas em JavaScript podem ser usadas para realizar verificações de erro (validação) na entrada do usuário, como uma string de data sendo formatada corretamente, ou um valor inserido (como uma idade) estando em um intervalo apropriado.

JavaScript pode até mesmo ser usada para modificar dinamicamente o código HTML sendo exibido. O navegador analisa o código HTML para uma estrutura de árvore na memória definida por um padrão chamado Document Object Model (DOM). O código JavaScript pode modificar a estrutura de árvore para executar certas operações. Por exemplo, suponha que um usuário precise inserir uma série de linhas de dados, como itens em uma conta sendo criada. Uma tabela contendo caixas de texto e outros métodos de entrada de formulário pode ser usada para reunir entrada do usuário. A tabela pode ter um tamanho padrão, mas, se mais linhas forem necessárias, o usuário pode clicar em um botão rotulado (por exemplo) "Adicionar item". Esse botão pode ser configurado para invocar uma função JavaScript que modifica a árvore DOM acrescentando uma linha extra na tabela.

Há também linguagens de scripting de uso especial para tarefas especializadas, como animação (por exemplo, Macromedia Flash e Shockwave) e modelagem tridimensional [Virtual Reality Markup Language (VRML)]. Linguagens de scripting também podem ser usadas no servidor, conforme veremos.

Servidores Web e sessões

Um **servidor Web** é um programa executado em uma máquina servidora que aceita solicitações de um navegador Web e envia de volta os resultados na forma de documentos HTML. O navegador e o servidor Web se comunicam por meio de um protocolo chamado **HyperText Transfer Protocol (HTTP)**. HTTP oferece recursos poderosos, além da simples transferência de documentos. O recurso mais importante é a capacidade de executar programas, com argumentos fornecidos pelo usuário, e entregar os resultados de volta como um documento HTML.

Como resultado, um servidor Web pode facilmente atuar como um intermediário para oferecer acesso a uma série de serviços de informação. Um novo serviço pode ser criado com a criação e instalação de um programa de aplicação que oferece o serviço. O padrão **Common Gateway Interface (CGI)** define como o servidor Web se comunica com os programas de aplicação. O programa de aplicação normalmente se comunica com um servidor de banco de dados, por meio de ODBC, JDBC ou outros protocolos, a fim de obter ou armazenar dados.

A Figura 8.4 mostra um serviço Web usando uma arquitetura de três camadas, com um servidor Web, um servidor de aplicação e um servidor de banco de dados. O uso de vários níveis de servidores aumenta o overhead do sistema; a interface CGI inicia um novo processo para atender cada solicitação, o que resulta em um overhead ainda maior.

A maioria dos serviços Web de hoje, portanto, utiliza uma arquitetura Web de duas camadas, em que o programa de aplicação é executado dentro do servidor Web, como na Figura 8.5. Estudamos sistemas com base na arquitetura de duas camadas com mais detalhes nas próximas subseções.

Não existe uma conexão contínua entre o cliente e o servidor Web; quando um servidor Web recebe uma solicitação, uma conexão é temporariamente criada para enviar a

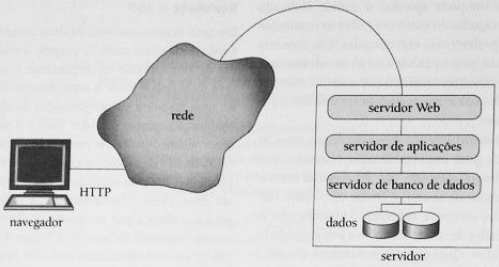


Figura 8.4 Arquitetura Web em três camadas.

solicitação e receber sua resposta. No entanto, a conexão é fechada e a próxima solicitação vem por uma nova conexão. Ao contrário, quando um usuário efetua o logon em um computador, ou se conecta a um banco de dados usando ODBC ou JDBC, uma sessão é criada, e as informações da sessão são retidas no servidor e no cliente até que a sessão termine – informações como o identificador do usuário e as opções de sessão que o usuário definiu. Um motivo importante para HTTP ser **sem conexão** é que a maioria dos computadores possui limites sobre o número de conexões simultâneas que eles podem acomodar, e se uma grande quantidade de sites abrisse conexões, esse limite seria excedido, negando o serviço para outros usuários. Com um serviço sem conexão, a conexão é partida assim que uma solicitação é satisfeita, deixando conexões disponíveis para outras solicitações.

No entanto, a maioria dos serviços de informação baseados na Web precisa da informação da sessão para permitir a interação significativa com o usuário. Por exemplo, os serviços normalmente restringem o acesso às informações, e por isso precisam autenticar usuários. A autenticação deve ser feita uma vez por sessão, e outras interações na sessão não devem exigir uma outra.

Para implementar sessões mesmo com as conexões sendo fechadas, informações extras precisam ser armazenadas no cliente e retornadas a cada solicitação em uma sessão. O servidor usa essa informação para identificar que uma solicitação faz parte de uma sessão do usuário. Informações extras sobre a sessão também precisam ser mantidas no servidor.

Essa informação extra normalmente é mantida na forma de um **cookie** no cliente; um cookie é simplesmente um pequeno texto contendo informações de identificação e com um nome associado. Por exemplo, `google.com` pode definir um cookie com o nome `prefs`, que codifica as preferências definidas pelo usuário, como a linguagem preferida e o número de respostas exibidas por página. A cada solicitação

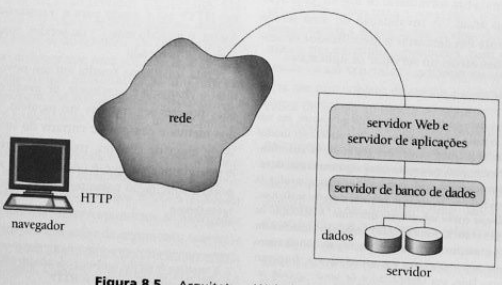


Figura 8.5 Arquitetura Web de duas camadas.

de busca, o `google.com` pode apanhar o cookie chamado `prefs` a partir do navegador do usuário e exibir os resultados de acordo com as preferências especificadas. Um domínio (site) tem permissão para apanhar apenas os cookies que ele definiu, e não os cookies definidos por outros domínios, e por isso nomes de cookie iguais podem ser utilizados por mais de um domínio.

Para fins de acompanhamento da sessão de um usuário, uma aplicação pode gerar um identificador de sessão (normalmente, um número aleatório não atualmente em uso como um identificador de sessão) e enviar um cookie chamado (por exemplo) `sessionid`, contendo o identificador de sessão. O identificador de sessão também é armazenado localmente no servidor. Quando uma solicitação chegar, o servidor de aplicação solicitará o cookie chamado `sessionid` do cliente. Se o cliente não tiver o cookie armazenado, ou se retornar um valor que não esteja atualmente registrado como um identificador de sessão válido no servidor, a aplicação conclui que a solicitação não faz parte de uma sessão atual. Se o valor do cookie combinar com um identificador de sessão armazenado, a solicitação é identificada como parte de uma sessão em andamento.

Se uma aplicação precisar identificar usuários com segurança, ela só poderá definir o cookie depois de autenticar o usuário; por exemplo, um usuário só pode ser autenticado quando um nome de usuário e senha válidos forem enviados.[1]

Para aplicações que não exigem alta segurança, como sites de notícias disponíveis publicamente, os cookies podem ser armazenados permanentemente no navegador e no servidor; eles identificam o usuário em visitas subseqüentes ao mesmo site, sem que qualquer informação de identificação seja digitada. Para aplicações que exigem maior segurança, o servidor pode invalidar (descartar) a sessão após um período de tempo limite, ou quando o usuário sair. (Normalmente, o usuário encerra as atividades pressionando um botão de sair, que envia um formulário de logout, cuja ação é invalidar a sessão atual.) A invalidação de uma sessão consiste simplesmente em descartar o identificador de sessão da lista de sessões ativas no servidor de aplicações.

1. O identificador de usuário pode ser armazenado no cliente, em um cookie chamado, por exemplo, `userid`. Esses cookies podem ser usados para aplicações de baixa segurança, como sites gratuitos, na identificação de seus usuários. Porém, para aplicações que exigem um nível de segurança mais alto, esse mecanismo cria um risco à segurança: o valor de um cookie pode ser mudado no navegador por um usuário malicioso, que pode então se disfarçar como um usuário diferente. A definição de um cookie (chamado `sessionid`, por exemplo) como um identificador de sessão gerado aleatoriamente (a partir de um espaço de números grande) torna altamente improvável que um usuário possa fingir ser outro usuário. Por outro lado, um identificador de sessão gerado sequencialmente seria suscetível ao disfarce.

Servlets e JSP

Em uma arquitetura Web de duas camadas, uma aplicação é executada como parte do próprio servidor Web. Uma forma de implementar tal arquitetura é carregar programas Java no servidor Web. A especificação de **servlet** Java define uma interface de programação de aplicação para a comunicação entre o servidor Web e o programa de aplicação. A classe `HttpServlet` em java implementa uma especificação de API de servlet; as classes de servlet usadas para implementar funções específicas são definidas como subclasses dessa classe.[2] Normalmente, a palavra *servlet* é usada para se referir a um programa (e classe) Java que implementa a interface de servlet. A Figura 8.6 mostra um exemplo de servlet; explicamos isso com detalhes mais adiante.

O código para um servlet (ou seja, um programa Java que implementa a interface de servlet) é carregado no servidor Web quando o servidor é iniciado, ou quando o servidor recebe uma solicitação HTTP remota para executar um servlet em particular. A tarefa de um servlet é processar tal solicitação, o que pode envolver o acesso a um banco de dados para apanhar informações necessárias, e gerar dinamicamente uma página HTML a ser retornada ao navegador cliente.

Um exemplo de servlet

Os servlets normalmente são usados para gerar dinamicamente respostas a solicitações HTTP. Eles podem acessar entradas fornecidas por meio de formulários HTML, aplicar "lógica de negócios" para decidir que resposta oferecer e depois gerar saída HTML para ser enviada de volta ao navegador.

A Figura 8.6 mostra um exemplo de código de servlet para implementar o formulário da Figura 8.2. O servlet se chama `BankQueryServlet`, porquanto o formulário especifica que `action="BankQuery"`. O servidor Web precisa ser informado de que esse servlet deve ser usado para tratar de solicitações para `BankQuery`. O formulário especifica que o mecanismo HTTP `get` é usado para a transmissão de parâmetros. Assim, o método `doGet( )` do servlet, conforme definido no código, é invocado.

Cada solicitação resulta em um novo thread dentro do qual a chamada é executada, de modo que várias solicitações possam ser tratadas em paralelo. Quaisquer valores dos menus e campos de entrada do formulário na página Web, além de cookies, passam por um objeto da classe `HttpServletRequest` que é criado para a solicitação, e a resposta à solicitação passa por um objeto da classe `HttpServletResponse`.

2. A interface de servlet também pode admitir solicitações não HTTP, embora nosso exemplo use apenas HTTP.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BankQueryServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        String type = request.getParameter("type");
        String number = request.getParameter("number");
        ... código para achar valor de empréstimo/saldo de conta ...
        ... usando JDBC para se comunicar com o banco de dados ..
        ... o valor está armazenado na variável balance

        result.setContentType("text/html");
        PrintWriter out = response.getWriter( );
        out.println("<HEAD><TITLE> Query Result</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("Balance on " + type + number + " = " + balance);
        out.println("</BODY>");
        out.close( );
    }
}
```

Figura 8.6 Exemplo de código de servet.

O método `doGet( )` no exemplo extrai valores de `type` e `number` dos parâmetros usando `request.getParameter( )`, e utiliza esses valores para executar uma consulta em um banco de dados. O código usado para acessar o banco de dados não aparece; consulte a seção "JDBC" do Capítulo 4 para obter detalhes de como usar o JDBC para acessar um banco de dados. O código do servlet retorna os resultados da consulta ao solicitante, imprimindo-os em formato HTML para o objeto `response` da classe `HttpServletResponse`.

Sessões de servlet

Lembre-se de que a interação entre um navegador e um servidor Web não tem estado. Ou seja, toda vez que o navegador faz uma solicitação ao servidor, o navegador precisa se conectar com o servidor, solicitar alguma informação, depois desconectar-se do servidor. Os cookies podem ser usados para reconhecer que uma solicitação é da mesma sessão do navegador que uma solicitação anterior. Porém, os cookies formam um mecanismo de baixo nível, e os programadores exigem uma abstração melhor para lidar com as sessões.

A API do servlet oferece um método para rastrear sessões e armazenar informações relacionadas à sessão. Invocar o método `getSession(false)` da classe `HttpServletRequest` obtém o objeto de `HttpSession` correspondente ao navegador que enviou a solicitação. Um valor de argumento `true` teria especificado que um novo objeto de sessão precisa ser criado se a solicitação for nova. Quando o método `getSession( )` é invocado, o servidor primeiro pede ao cliente para retornar um cookie com um nome especificado.

Se o cliente não tiver um cookie com esse nome, ou retornar um valor que não combina com qualquer sessão em andamento, então a solicitação não faz parte de uma sessão em andamento. Nesse caso, o servlet poderia direcionar o usuário a uma página de login, que poderia permitir ao usuário fornecer um nome de usuário e senha. O servlet correspondente à página de login poderia verificar se a senha corresponde ao usuário (por exemplo, examinando informações de autenticação no banco de dados). Outros métodos para autenticar usuários são possíveis. Se o usuário for corretamente autenticado, o servlet de login executará um `getSession(true)`, o qual retornará um novo objeto de sessão. Para criar uma nova sessão, o servidor Web internamente executa as seguintes tarefas: define um cookie (chamado, por exemplo, sessionID), com um identificador de sessão como seu valor associado no navegador do cliente, cria um novo objeto de sessão e associa o valor do identificador de sessão ao objeto de sessão.

O código do servlet também pode armazenar e consultar pares (nome de atributo, valor) no objeto `HttpSession`, para manter o estado por várias solicitações dentro de uma sessão. Por exemplo, o servlet de login poderia armazenar a identificação do usuário como um parâmetro de sessão executando o método `session.setAttribute("userid", userid)` no objeto de sessão (em que a variável Java `userid` contém o identificador do usuário), após o usuário ser autenticado e o objeto de sessão ter sido criado.

Se a solicitação foi parte de uma sessão em andamento, o navegador retorna o valor do cookie, e o objeto de sessão correspondente é retornado pelo servidor Web. O servlet pode então apanhar parâmetros de sessão, como userid, do objeto de sessão executando o método `session.getAttribute("userid")`. Se o atributo `userid` não estivesse definido, a função retornaria um valor nulo, que indicaria que o usuário cliente não foi autenticado.

Ciclo de vida do servlet

O ciclo de vida de um servlet é controlado pelo servidor Web em que ele foi implantado. Quando existe uma solicitação do cliente para um servlet específico, o servidor primeiro verifica se uma instância do servlet existe ou não. Se não, o servidor Web carrega servlet na máquina virtual Java (JVM) e cria uma instância da classe servlet. Além disso, o servidor chama o método `init( )` para inicializar a instância do servlet. Observe que cada instância do servlet só é inicializada quando carregada.

Depois de certificar-se de que a instância do servlet existe, o servidor invoca o método `service` do servlet, com um objeto `request` e um objeto `response` como parâmetros. Como padrão, o servidor cria um novo thread para executar o método `service`. Assim, várias solicitações em um servlet podem ser executadas em paralelo, sem ter de esperar solicitações anteriores para completar a execução. O método `service` chama `doGet` ou `doPost` conforme a necessidade.

Quando não for mais necessário, um servlet pode ser encerrado chamando o método `destroy( )`. O servidor pode ser configurado para encerrar automaticamente um servlet se nenhuma solicitação tiver sido feita em um servlet dentro de um período de tempo limite; o período de tempo limite é um parâmetro do servidor que pode ser definido conforme a necessidade da aplicação.

Suporte para servlet

O programa Java `servletrunner` oferece suporte mínimo para executar servlets e é uma forma rápida de começar a usar servlets. Ele pode receber solicitações em uma porta especificada, invocar servlets apropriados e enviar a resposta de volta ao cliente.

Muitos servidores de aplicação oferecem suporte interno para servlets. Alguns exemplos incluem o servidor de aplicações de fonte aberto JBoss, Java Web Server da Sun e WebSphere Application Server da IBM. Um dos mecanismos de servlet independentes mais populares é o Tomcat Server, do Apache Jakarta Project, que é um projeto de fonte aberto gratuito.

Esses servidores de aplicações oferecem diversos outros serviços, além do suporte básico para servlet fornecido pelo *servletrunner*. Por exemplo, se um código de servlet for modificado, eles podem detectar isso e recompilar e recarregar o servlet de forma transparente. Como outro exemplo, muitos servidores de aplicação permitem que o servidor seja executado em várias máquinas, para melhorar o desempenho e rotear as solicitações para uma cópia apropriada. Eles também oferecem funcionalidade para monitorar o status do servidor de aplicações, incluindo estatísticas de desempenho. Muitos servidores de aplicação também têm suporte para a plataforma Java 2 Enterprise Edition (J2EE), que oferece suporte e APIs para uma série de tarefas, como para tratamento de objetos, processamento paralelo por vários servidores de aplicação e tratamento de dados XML (XML é descrita mais adiante no Capítulo 10).

Scripting no lado servidor

A escrita até mesmo de uma aplicação Web simples em uma linguagem de programação como Java ou C é uma tarefa demorada, que exige muitas linhas de código e programadores acostumados com os detalhes da linguagem. Uma técnica alternativa, a de **scripting no lado servidor**, oferece um método muito mais fácil para criar muitas aplicações. As linguagens de scripting oferecem construções que podem ser embutidas dentro de documentos HTML. No scripting no servidor, antes do envio de uma página Web, o servidor executa os scripts embutidos dentro do conteúdo HTML da página. Cada parte do sc, quando executada, pode gerar texto acrescentado à página (ou pode até mesmo excluir o conteúdo da página). O código fonte dos scripts é removido da página, para que o cliente nem saiba que a página originalmente continha algum código. O script executado pode conter código SQL que é executado em um servidor de banco de dados.

Várias linguagens de scripting apareceram nos últimos anos, incluindo Server-Side JavaScript da Netscape, JScript da Microsoft, Java Server Pages (JSP) da Sun, o PHP Hypertext Preprocessor (PHP), ColdFusion Markup Language (CFML) da ColdFusion e DTML da Zope. Na verdade é possível até mesmo incorporar código escrito em linguagens de scripting mais antigas, como VBScript, Perl e Python, em páginas HTML. Por exemplo, Active Server Pages (ASP) da Microsoft aceita VBScript e JScript embutida. Essas linguagens têm suporte para recursos semelhantes, mas diferem no estilo de programação e na facilidade com que aplicações simples podem ser criadas.

A seguir, descrevemos rapidamente a **Java Server Pages (JSP)**, uma linguagem de scripting que permite que os programadores HTML misturem HTML estática com HTML gerada dinamicamente. A motivação é que, para muitas páginas Web dinâmicas, a maior parte do conteúdo ainda é estático (ou seja, o mesmo conteúdo está presente sempre

que a página for gerada). O conteúdo dinâmico das páginas Web (que são geradas, por exemplo, com base nos parâmetros do formulário) normalmente é uma pequena parte da página. A criação de tais páginas com a escrita de código de servlet resulta em uma grande quantidade de HTML sendo codificada como strings Java. JSP, em vez disso, permite que o código Java seja embutido na HTML estática; o código Java embutido gera a parte dinâmica da página. Os scripts JSP normalmente são traduzidos para o código de servlet, que é então compilado, mas o programador de aplicação não tem o trabalho de escrever grande parte do código Java para criar o servlet.

A Figura 8.7 mostra o texto fonte de uma página HTML que inclui um script JSP. O código Java é destacado do código HTML ao seu redor porque é delimitado com <% ... %>. O script chama o `request.getParameter( )` para obter o valor do atributo `name`. Dependendo do valor, o script decide o que deve ser impresso após "Hello". Um exemplo mais real pode realizar ações mais complexas, como pesquisar valores de um banco de dados usando JDBC.

JSP também admite o conceito de uma *biblioteca de tags*, que permite o uso de tags que são muito semelhantes às tags HTML, mas são interpretadas no servidor, sendo replicadas pelo código HTML gerado corretamente. JSP oferece um conjunto padrão de tags que definem variáveis e fluxo de controle (repetidores, if-then-else), junto com uma linguagem de expressão baseada em JavaScript (mas interpretada no servidor). O conjunto de tags é extensível, e diversas bibliotecas de tags foram implementadas. Por exemplo, existe uma biblioteca de tags que admite a exibição paginada de grandes conjuntos de dados, além de uma biblioteca que simplifica a exibição e a análise de datas e horas. Consulte as notas bibliográficas para obter mais informações sobre bibliotecas de tags JSP.

Montando grandes aplicações Web

Na montagem de aplicações Web, grande parte do esforço de programação se encontra na interface com o usuário e não nas tarefas relacionadas ao banco de dados. Na primeira parte desta seção, estudamos maneiras de reduzir o esforço de programação para essa tarefa. Mais adiante nesta seção, descrevemos algumas técnicas para melhorar o desempenho da aplicação.

Construindo interfaces Web

Descrevemos, a seguir, várias técnicas para reduzir o esforço de programação na montagem da interface com o usuário.

Muitas construções HTML são mais bem geradas pelo uso de funções Java definidas corretamente, em vez de serem escritas como parte do código de cada página Web. Por exemplo, os formulários de endereço normalmente exigem um menu contendo nomes de país ou estado. Em vez de escrever código HTML extenso para criar o menu necessário sempre que ele for usado, é preferível definir uma função que emite o menu e chamá-la sempre que for necessária.

Os menus normalmente são gerados melhor a partir dos dados em um banco de dados, como uma tabela contendo nomes de países ou nomes de estados. A função que gera o menu executa uma consulta de banco de dados e preenche o menu, usando o resultado da consulta. A inclusão de um país ou estado, então, exige apenas uma mudança no banco de dados, e não no código da aplicação. Essa técnica tem a desvantagem em potencial de exigir maior interação com o banco de dados, mas esses extras podem ser reduzidos com o caching dos resultados de consulta no servidor de aplicações.

Formulários para entrada de datas e horas, ou entradas que exigem validação, da mesma forma, são gerados melhor pela chamada de funções definidas corretamente. Essas funções podem enviar código JavaScript para realizar a validação no navegador.

A exibição de um conjunto de resultados a partir de uma consulta é uma tarefa comum para muitas aplicações de banco de dados. É possível montar uma função genérica que obtém uma consulta SQL (ou `ResultSet`) como argumento e exibe as tuplas no resultado da consulta (ou Re-

```
<html>
<head> <title> Hello  </title> </head>
<body>
<H1>
   < % if (request.getParameter("name") == null)
    { out.println("Hello World"); }
    else { out.println("Hello, " + request.getParameter("name")); }
   %>
</H1>
</body>
</html>
```

Figura 8.7 Um texto fonte HTML com um script JSP.

sultSet) em um formato tabular. Chamadas de metadados JDBC podem ser usadas para localizar informações como o número de colunas e o nome e os tipos das colunas no resultado da consulta; essa informação é utilizada para exibir o resultado da consulta.

Para lidar com situações em que o resultado da consulta é muito grande, essa função de exibição de resultado da consulta pode providenciar *paginação* de resultados. A função pode exibir um número fixo de registros em uma página e oferecer controles para pular para a página seguinte ou anterior, ou saltar para determinada página dos resultados.

Infelizmente, não existe uma API Java padrão (usada por todos) para as funções executarem as tarefas de interface com o usuário descritas anteriormente. A criação de tal biblioteca pode ser um projeto de programação interessante.

Microsoft Active Server Pages

Active Server Pages (ASP) da Microsoft, e sua versão mais recente, Active Server Pages.NET (ASP.NET), é uma alternativa bastante usada a JSP/Java. ASP.NET é semelhante a JSP, porque o código em uma linguagem como Visual Basic ou C# pode ser embutido dentro do código HTML. Além disso, ASP.NET oferece uma série de controles (comandos de scripting) que são interpretados no servidor, e gera HTML que é enviada ao cliente. Esses controles podem simplificar bastante a construção de interfaces Web. Oferecemos uma rápida visão geral dos benefícios que esses controles oferecem.

Por exemplo, controles como menus suspensos e caixas de listagem podem ser associados a um objeto `DataSet`. O objeto `DataSet` é semelhante a um objeto `ResultSet` do JDBC, e normalmente é criado pela execução de uma consulta no banco de dados. O conteúdo do menu HTML é então gerado a partir do conteúdo do objeto `DataSet`; por exemplo, uma consulta pode colocar os nomes de todos os departamentos de uma organização em um `DataSet`, e o menu associado teria esses nomes. Assim, os menus que dependem do conteúdo do banco de dados podem ser criados de uma maneira conveniente com muito pouca programação.

Controles de validação podem ser acrescentados para formar campos de entrada; estes especificam, de forma declarativa, restrições de validade como intervalos de valor, ou se a entrada é uma entrada obrigatória para a qual um valor precisa ser fornecido pelo usuário. O servidor cria o código HTML apropriado combinado com JavaScript para realizar a validação no navegador do usuário. As mensagens de erro a serem exibidas na entrada inválida podem ser associadas a cada controle de validação.

Ações do usuário podem ser especificadas para ter uma ação associada no servidor. Por exemplo, um controle de menu pode especificar que a seleção de um valor a partir de um menu tem uma ação associada no servidor (isso é implementado pelo código JavaScript gerado pelo servidor). O código Visual Basic/C# que exibe os dados pertencentes ao valor selecionado pode ser associado à ação no servidor. Assim, a seleção de um valor em um menu pode resultar na atualização dos dados associados na página, sem exigir que o usuário clique em um botão submit.

O controle `DataGrid` oferece um modo bastante conveniente de exibir resultados de consulta. Um `DataGrid` é associado a um objeto `DataSet`, que normalmente é o resultado de uma consulta. O servidor gera código HTML que exibe o resultado da consulta como uma tabela. Os cabeçalhos de coluna são gerados automaticamente a partir dos metadados de resultado da consulta. Além disso, `DataGrids` oferecem vários recursos, como paginação, e permitem que o usuário classifique o resultado sobre as colunas escolhidas. Todo o código HTML, além da funcionalidade no servidor para implementar esses recursos, é gerado automaticamente pelo servidor. O `DataGrid` até mesmo permite que os usuários editem os dados e submetam mudanças ao servidor. O desenvolvedor da aplicação pode especificar uma função, a ser executada quando uma linha for editada, que pode realizar a atualização no banco de dados.

O Microsoft Visual Studio oferece uma interface gráfica com o usuário para a criação de páginas ASP usando esses recursos, além de reduzir o esforço de programação.

Veja, nas notas bibliográficas, referências a mais informações sobre ASP.NET.

Melhorando o desempenho da aplicação

Os sites podem ser acessados por milhões de pessoas do mundo inteiro, a milhares de solicitações por segundo, ou mais ainda, para os sites mais populares. Garantir que as solicitações sejam atendidas com tempos de resposta baixos é um desafio importante para os desenvolvedores de sites.

Técnicas de caching de vários tipos são usadas para explorar semelhanças entre as transações. Por exemplo, suponha que o código da aplicação para atender cada solicitação do usuário precise contatar um banco de dados por meio do JDBC. A criação de uma nova conexão JDBC pode levar vários milissegundos, de modo que a abertura de uma nova conexão para cada solicitação do usuário não é uma boa ideia se velocidades de transação muito altas tiverem de ser admitidas.

O **pooling de conexões** é usado para reduzir essa sobrecarga; ele funciona da seguinte maneira. O gerenciador do pool de conexões (uma parte do servidor de aplicações) cria um pool (ou seja, um conjunto) das conexões ODBC/JDBC abertas. Em vez de abrir uma nova conexão com o banco de dados, o código atendendo uma solicitação do usuário (normalmente, um servlet) pede (solicita) uma

conexão do pool de conexões e retorna a conexão ao pool quando o código (servlet) conclui seu processamento. Se o pool não tiver conexões livres quando uma conexão for solicitada, uma nova conexão é aberta com o banco de dados (cuidado para não exceder o número máximo de conexões que o sistema de banco de dados pode admitir simultaneamente). Se houver muitas conexões abertas que não foram usadas por algum tempo, o gerenciador do pool de conexões poderá fechar algumas. Muitos servidores de aplicação, e os drivers ODBC/JDBC mais novos, oferecem um gerenciador de pool de conexão embutido.

Um erro comum que muitos programadores cometem ao criar aplicações Web é esquecer de fechar uma conexão JDBC aberta (ou, de forma equivalente, quando o pooling de conexão é usado, esquecer de retornar a conexão ao pool de conexão). Cada solicitação, então, abre uma nova conexão com o banco de dados, e o banco de dados logo alcança o limite de quantas conexões abertas ele pode ter de uma só vez. Esses problemas normalmente não aparecem no teste em pequena escala, pois os bancos de dados normalmente permitem centenas de conexões abertas, mas aparecem apenas no uso intensivo. Alguns programadores assumem que as conexões, como a memória alocada pelos programas Java, têm coleta de lixo automática. Infelizmente, isso não acontece, e os programadores são responsáveis por fechar as conexões que foram abertas.

Certas solicitações podem resultar em exatamente a mesma consulta sendo outra vez submetida ao banco de dados. O custo da comunicação com o banco de dados pode ser bastante reduzido pelo caching de resultados das consultas anteriores e sua reutilização, desde que o resultado da consulta não tenha mudado no banco de dados. Alguns servidores Web admitem esse caching de resultado da consulta.

Os custos podem ser reduzidos ainda mais pelo caching da página Web final que é enviada em resposta a uma solicitação. Se uma nova solicitação vier com exatamente os mesmos parâmetros que uma solicitação anterior e se a página Web resultante estiver no cache, então ela pode ser reutilizada para evitar o custo de recalcular a página. O caching pode ser feito no nível de fragmentos de páginas Web, que depois são montadas para criar páginas Web completas.

Os resultados de consulta em cache e páginas Web em cache são formas de views materializadas. Se os dados básicos do banco de dados mudarem, eles poderão ser descartados, recalculados ou ainda atualizados de forma incremental, como na manutenção de view materializada (descrita mais adiante, na seção "Views materializadas" do Capítulo 14). Alguns sistemas de banco de dados (como Microsoft SQL Server) oferecem um modo de o servidor de aplicação registrar uma consulta com o banco de dados e obter uma **notificação** do banco de dados quando o resultado da consulta mudar. Esse mecanismo de notificação pode ser usado para garantir que os resultados da consulta em cache no servidor de aplicações estejam atualizados.

Triggers

Um **trigger** é uma instrução que o sistema executa automaticamente como um efeito colateral de uma modificação no banco de dados. Para criar um mecanismo de trigger, temos de cumprir dois requisitos:

1. Especificar quando um trigger deve ser executado. Isso é desmembrado em um *evento* que faz com que o trigger seja verificado e uma *condição* que precisa ser satisfeita para que a execução do trigger prossiga.
2. Especificar as *ações* a serem tomadas quando o trigger for executado.

Esse modelo de trigger é conhecido como **modelo evento-condição-ação** para triggers.

O banco de dados armazena triggers como se fossem dados normais, de modo que sejam persistentes e acessíveis a todas as operações de banco de dados. Quando entramos com um trigger no banco de dados, o sistema de banco de dados assume a responsabilidade por executá-lo sempre que o evento especificado ocorre e a condição correspondente é satisfeita.

Necessidade de triggers

Triggers são mecanismos úteis para alertar as pessoas ou para iniciar certas tarefas automaticamente quando certas condições são atendidas. Como ilustração, suponha que, em vez de permitir saldos de conta negativos, o banco trate de saldos devedores definindo o saldo da conta como zero e criando um empréstimo no valor do saldo devedor. O banco dá a esse empréstimo um número idêntico ao número da conta com saldo devedor. Nesse exemplo, a condição para executar o trigger é uma atualização na relação *conta* que resulte em um valor negativo de *saldo*. Suponha que o saque de algum dinheiro na conta de Jones tenha tornado o saldo da conta negativo. Imagine que t indica a tupla de conta com um valor de *saldo* negativo. As ações a serem tomadas são:

- Inserir uma nova tupla s na relação *empréstimo* com

$$s[número\_empréstimo] = t[número\_conta]$$
$$s[nome\_agência] = t[nome\_agência]$$
$$s[quantia] = -t[saldo]$$

(Observe que, como t[saldo] é negativo, negamos t[saldo] para obter o valor do empréstimo – um número positivo.)

- Inserir uma nova tupla u na relação tomador com

  $u[nome\_cliente] = \text{"Jones"}$
  $u[numero\_empréstimo] = t[numero\_conta]$

- Definir $t[saldo]$ como 0.

Como outro exemplo do uso de triggers, suponha que um depósito queira manter um estoque mínimo de cada item: quando o nível de estoque de um item ficar abaixo do nível mínimo, um pedido deverá ser feito automaticamente. É assim que a regra comercial pode ser implementada por meio de triggers: em uma atualização do nível de estoque de um item, o trigger deve compará-lo ao nível de estoque mínimo e, se estiver abaixo ou igual ao mínimo, um novo pedido é acrescentado a uma relação *pedidos*.

Observe que os sistemas de trigger normalmente não podem realizar atualizações fora do banco de dados e, portanto, no exemplo de reposição de estoque, não podemos usar um trigger para fazer um pedido diretamente ao mundo exterior. Em vez disso, acrescentamos um pedido à relação *pedidos*, como no exemplo de estoque. É preciso criar um processo do sistema em execução permanente, que periodicamente varre a relação *pedidos* e faz os pedidos. Esse processo do sistema também anota quais tuplas na relação *pedidos* foram processadas e quando cada pedido foi feito. O processo também acompanharia as entregas dos pedidos e alertaria gerentes em caso de condições excepcionais, como atrasos nas entregas. Alguns sistemas de banco de dados oferecem suporte interno para enviar e-mail a partir de consultas SQL e triggers, usando a técnica que indicamos.

Triggers em SQL

Os sistemas de banco de dados baseados em SQL utilizam bastante os triggers, embora antes da SQL:1999 eles não fizessem parte do padrão SQL. Infelizmente, cada sistema de banco de dados implementou sua própria sintaxe para ostriggers, levando a incompatibilidades. Esboçamos na Figura 8.8 a sintaxe da SQL:1999 para os triggers (que é semelhante à sintaxe nos sistemas de banco de dados IBM DB2 e Oracle).

Essa definição de trigger especifica que o trigger é iniciado *após* qualquer atualização da relação *conta* ser executada. Uma instrução de atualização SQL poderia atualizar várias tuplas da relação, e a cláusula **for each row** no código do trigger percorreria explicitamente cada linha atualizada. A cláusula **referencing new row as** cria uma variável *nrow* (chamada **variável de transição**), que armazena o valor de uma linha atualizada após a atualização.

A instrução **when** especifica uma condição, a saber, $nrow.saldo < 0$. O sistema executa o restante do corpo do trigger somente para tuplas que satisfazem a condição. A cláusula **begin atomic** ... **end** serve para coletar várias instruções SQL em uma única instrução composta. As duas instruções **insert** com a estrutura **begin** ... **end** executam tarefas específicas de criação de novas tuplas nas relações *tomador* e *empréstimo* para representar o novo empréstimo. A instrução **update** serve para definir o saldo da conta de volta a 0 a partir do seu antigo valor negativo.

O evento de disparo e as ações podem tomar muitas formas:

- O *evento* de disparo pode ser **insert** ou **delete**, em vez de **update**.

  Por exemplo, a ação sobre **delete** de uma conta poderia ser verificar se os mantenedores da conta possuem

```
create trigger trigger_saldo_devedor after update on conta
referencing new row as nrow
for each row
when nrow.saldo < 0
begin atomic
  insert into tomador
    (select nome_cliente, numero_conta
     from depositante
     where nrow.numero_conta=depositante.numero_conta);
  insert into empréstimo values
    (nrow.numero_conta, nrow.nome_agência, – nrow.saldo);
  update conta set saldo = 0
    where conta.numero_conta = nrow.numero_conta
end
```

Figura 8.8 Exemplo de sintaxe SQL:1999 para os triggers.

quaisquer contas restantes, e se não tiverem, excluí-los da relação *depositante*. Você pode definir esse trigger como um exercício (Exercício prático 8.5).

Como outro exemplo, se um novo *depositante* for inserido, a ação disparada poderia ser enviar uma carta de boas-vindas ao depositante. Obviamente, um trigger não pode causar tal ação diretamente fora do banco de dados, mas poderia acrescentar uma tupla a uma relação que armazena endereços aos quais as cartas de boas-vindas precisam ser enviadas. Um processo separado passaria por essa tabela, e imprimiria cartas a serem enviadas.

Muitos sistemas de banco de dados dão suporte a uma série de outros eventos de disparo, por exemplo, quando um usuário (aplicação) efetua o login no banco de dados (ou seja, abre uma conexão), quando o sistema desliga ou quando são feitas mudanças nas configurações do sistema.

- Para as atualizações, o trigger pode especificar colunas cuja atualização faz com que o trigger seja executado. Por exemplo, se a primeira linha do trigger de saldo devedor fosse substituída por

```
create trigger trigger_saldo_devedor after update of
    saldo on conta
```

então o trigger seria executado somente nas atualizações de *saldo*; as atualizações de outros atributos não causariam sua execução.

- A cláusula referencing old row as pode ser usada para criar uma variável armazenando o valor antigo de uma linha atualizada ou excluída. A cláusula **referencing new row as** pode ser usada com inserções além de atualizações.
- Triggers podem ser ativados antes (**before**) do evento (insert/delete/update), ao invés de depois (**after**) do evento.

Esses triggers servem como restrições extras que podem impedir atualizações inválidas. Por exemplo, se não quisermos permitir saldos devedores, podemos criar um trigger **before** que verifica se o novo saldo é negativo e, se for, reverte a transação. Embora um trigger **after** pudesse ter sido usado para essa finalidade, seu uso resultaria na atualização sendo feita primeiro, para depois a transação ser revertida.

Como outro exemplo, suponha que o valor em um campo de número de telefone de uma tupla inserida esteja em branco, o que indica a ausência de um número de telefone. Podemos definir um trigger que substitua o valor pelo valor **null**. A instrução **set** pode ser usada para executar tais modificações.

```
create trigger trigger_nulo before update on r
referencing new row as nrow
for each row
when nrow.número_telefone = ''
set nrow.número_telefone = null
```

- Em vez de executar uma ação para cada linha afetada, podemos executar uma única ação para a instrução SQL inteira que causou o insert/delete/update. Para fazer isso, usamos a cláusula **for each statement** no lugar da cláusula **for each row**.

As cláusulas **referencing old table as** ou **referencing new table as** podem ser usadas para referenciar tabelas temporárias (chamadas *tabelas de transição*) contendo todas as linhas afetadas. As tabelas de transição não podem ser usadas com triggers **before**, mas podem ser usadas com triggers **after**, independentemente de serem triggers de instrução ou triggers de linha.

Uma única instrução SQL pode, então, ser usada para executar várias ações com base nas tabelas de transição.

- Triggers podem ser desativados ou ativados; como padrão, eles são ativados quando criados, mas podem ser desativados por meio de **alter trigger** *nome_trigger* **disable** (alguns bancos de dados utilizam uma sintaxe alternativa, como **disable trigger** *nome_trigger*). Um trigger que foi desativado pode ser ativado novamente. E também pode ser descartado usando o comando **drop trigger** *nome_trigger*, que o remove permanentemente,.

Retornando ao nosso exemplo de estoque de depósito, suponha que tenhamos as seguintes relações:

- *estoque(item, nível)*, que observa a quantidade atual (número/peso/volume) do item no depósito
- *nívelmín(item, nível)*, que observa a quantidade mínima que deve ser mantida para esse item
- *novopedido(item, quantidade)*, que observa a quantidade do item a ser pedida quando seu nível ficar abaixo do mínimo
- *pedidos(item, quantidade)*, que observa a quantidade do item a ser pedida.

Observe que precisamos ter cuidado para fazer um pedido apenas quando a quantidade cair de acima do nível mínimo para abaixo do nível mínimo. Se verificar apenas se o novo valor após uma atualização está abaixo do nível mínimo, poderemos fazer um pedido erroneamente quando o item já tiver sido pedido. Podemos então usar o trigger mostrado na Figura 8.9 para pedir o item.

Muitos sistemas de banco de dados oferecem implementações de trigger fora do padrão, ou implementam apenas alguns dos recursos de trigger. Por exemplo, muitos siste-

```
create trigger trigger_pedido after update of quantia on estoque
referencing old row as orow, new row as nrow
for each row
when nrow.nivel <= (select nivel
                    from minnivel
                    where minnivel.item = orow.item)
and orow.nivel > (select nivel
                  from minnivel
                  where minnivel.item = orow.item)
begin
  insert into pedidos
    (select item, quantia
     from novopedido
     where novopedido.item=orow.item)
end
```

Figura 8.9 Exemplo de trigger para novo pedido de um item.

mas de banco de dados não implementam a cláusula **before**, e a palavra-chave **on** é usada no lugar de **after**. Eles podem não implementar a cláusula **referencing**. Em vez disso, podem especificar tabelas de transição usando as palavras-chave **inserted** ou **deleted**. A Figura 8.10 ilustra como o trigger de saldo devedor seria escrito no Microsoft SQL Server. Leia o manual do usuário do sistema de banco de dados que você usa a fim de obter mais informações sobre os recursos do trigger que ele admite.

Quando não usar triggers

Existem muitos bons usos para triggers, como aqueles que já vimos na seção anterior, mas alguns usos podem ser tratados melhor por meio de técnicas alternativas. Por exemplo, no passado, os projetistas de sistemas usavam triggers para manter dados de resumo. Por exemplo, eles usavam triggers no insert/delete/update de uma relação *funcionário* contendo atributos *salário* e *departamento* para manter o total de salários de cada departamento. Porém, muitos sistemas de banco de dados hoje admitem views materializadas (ver seção "Definição de view" do Capítulo 3), que oferecem um modo muito mais fácil de manter dados de resumo. Os projetistas também usavam muito triggers para replicar bancos de dados; eles usavam triggers no insert/delete/update de cada relação para registrar as mudanças nas relações, chamadas relações **change** ou **delta**. Um processo separado copiava as mudanças para a réplica (cópia) do

```
create trigger trigger_saldo_devedor on conta
for update
as
if inserted.saldo < 0
begin
 insert into tomador
    (select nome_cliente, número_conta
     from depositante, inserted
     where inserted.número_conta=depositante.número_conta)
 insert into empréstimo values
    (inserted.número_conta, inserted.nome_agência, – inserted.saldo)
 update conta set saldo = 0
    from conta, inserted
    where conta.número_conta = inserted.número_conta
end
```

Figura 8.10 Exemplo de trigger na sintaxe do MS-SQL Server.

banco de dados, e o sistema executava as mudanças na réplica. Porém, os sistemas de banco de dados modernos oferecem facilidades embutidas para a replicação de banco de dados, tornando os triggers desnecessários para a replicação na maioria dos casos.

De fato, muitas aplicações de trigger, incluindo nosso trigger de exemplo de saldo devedor, podem ser substituídas pelo uso apropriado dos procedimentos armazenados. Por exemplo, suponha que as atualizações no atributo *saldo de conta* sejam feitas apenas por meio de um procedimento armazenado em particular. Esse procedimento, por sua vez, verificaria o saldo negativo e executaria as ações do trigger de saldo devedor. Os programadores devem ter cuidado para não atualizar diretamente o valor de *saldo*, mas atualizá-lo apenas por meio do procedimento armazenado; isso poderia ser garantido não dando a aplicação/usuário a autorização para atualizar o atributo *saldo*, mas oferecendo autorização para executar o procedimento armazenado que está associado. Um encapsulamento semelhante pode ser usado para substituir o trigger *novopedido* por um procedimento armazenado.

Outro problema com os triggers está na execução não intencionada da ação disparada quando os dados são carregados de uma cópia de backup, ou quando atualizações do banco de dados em um site são replicadas em um site de backup. Nesses casos, a ação disparada já foi executada, e normalmente não deve ser executada novamente. Ao carregar dados, os triggers podem ser desativados explicitamente. Para sistemas de réplica de backup que podem ter de assumir o comando no lugar do sistema principal, os triggers teriam de ser desativados inicialmente e ativados quando o site de backup assumisse o processamento no lugar do principal. Como alternativa, alguns sistemas de banco de dados permitem que os triggers sejam especificados como **not for replication**, o que garante que eles não sejam executados no site de backup durante a replicação do banco de dados. Outros sistemas de banco de dados oferecem uma variável do sistema que indica que o banco de dados é uma réplica na qual as ações do banco de dados estão sendo reproduzidas; o corpo do trigger deve verificar essa variável e sair se ela for verdadeira. As duas soluções evitam a necessidade de desativação e ativação explícita de triggers.

Os triggers devem ser escritos com muito cuidado, pois um erro de trigger detectado em tempo de execução causa a falha da instrução insert/delete/update que disparou o trigger. Além do mais, a ação de um trigger pode disparar um outro. No pior dos casos, isso poderia levar a uma cadeia infinita de disparos. Por exemplo, suponha que um trigger insert sobre uma relação tenha uma ação que causa outro insert (novo) na mesma relação. A ação de insert então dispara outra ação de insert, e assim por diante, *ad infinitum*. Os sistemas de banco de dados normalmente limitam a extensão de tais cadeias de triggers (por exemplo, a 16 ou 32) e consideram que existe um erro se houver cadeias de disparo maiores.

Ocasionalmente, os triggers são chamados de *regras*, ou *regras ativas*, mas não devem ser confundidos com a regras do Datalog (ver seção "Datalog" do Capítulo 5), que na realidade são definições de view.

Autorização em SQL

Vimos o conjunto básico de privilégios da SQL na seção "Autorização" do Capítulo 4, incluindo os privilégios delete, insert, select e update.

Além dessas formas de privilégios para acesso aos dados, podemos (conceitualmente) conceder a um usuário diferentes tipos de autorização para modificar o esquema do banco de dados:

- Autorização para criar novas relações
- Autorização para acrescentar atributos ou excluir atributos de uma relação
- Autorização para descartar uma relação

O padrão SQL especifica um mecanismo de autorização primitivo para o esquema do banco de dados. Somente o proprietário do esquema pode executar qualquer modificação no esquema. Assim, as modificações no esquema – como a criação ou exclusão de relações, a inclusão ou o descarte de atributos de relações, e a inclusão ou o descarte de índices – podem ser executadas somente pelo proprietário do esquema. Várias implementações de banco de dados possuem mecanismos de autorização mais poderosos para esquemas de banco de dados, semelhante àqueles discutidos anteriormente, mas esses mecanismos são fora do padrão.

A SQL também inclui um privilégio **references**, que permite que um usuário declare chaves estrangeiras ao criar relações. Inicialmente, pode parecer que não existe motivo para impedir que os usuários criem chaves estrangeiras referenciando outra relação. Porém, lembre-se de que as restrições de chave estrangeira restringem as operações de exclusão e atualização sobre a relação referenciada. Suponha que U_1 crie uma chave estrangeira em uma relação r referenciando o atributo *nome_agência* da relação *agência* e depois insira uma tupla em r pertencente à agência Perryridge. Não é mais possível excluir a agência Perryridge da relação *agência* sem também modificar a relação r. Assim, a definição de uma chave estrangeira por U_1 restringe a atividade futura por outros usuários; portanto, há necessidade do privilégio **references**.

O privilégio **references** sobre s também é necessário para criar uma restrição **check** sobre uma relação r se a restrição tiver uma subconsulta referenciando a relação s.

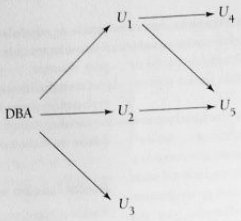


Figura 8.11 Gráfico de concessão de autorização.

A SQL define um privilégio **execute** que autoriza um usuário a executar uma função ou procedimento. Assim, somente um usuário que precise do privilegio execute sobre uma função *f*( ) pode chamá-la (seja diretamente ou de dentro de uma consulta SQL).

A SQL também inclui um privilégio **usage** que autoriza um usuário a usar um domínio especificado (lembre-se de que um domínio corresponde à noção de um tipo na linguagem de programação, e pode ser definido pelo usuário).

A forma definitiva de autoridade é aquela dada ao administrador de banco de dados. O administrador de banco de dados pode autorizar novos usuários, reestruturar o banco de dados e assim por diante. Essa forma de autorização é semelhante à de um **superuser** ou operador para um sistema operacional.

Concessão de privilégios

Um usuário que recebeu alguma forma de autorização pode ter permissão para passar adiante essa autorização a outros usuários. Porém, temos de ter o cuidado de como a autorização pode ser passada entre os usuários, para garantir que ela possa ser revogada em algum momento no futuro.

Considere, como um exemplo, a concessão da autorização de atualização sobre a relação *empréstimo* do banco de dados bancário. Considere que, inicialmente, o administrador de banco de dados conceda autorização de atualização sobre *empréstimo* para os usuários U_1, U_2 e U_3, que, por sua vez, podem passar essa autorização adiante para outros usuários. A passagem de autorização de um usuário para outro pode ser representada por um **gráfico de autorização**. Os nós nesse gráfico são os usuários. O gráfico inclui uma aresta $U_i \rightarrow U_j$ se o usuário U_i conceder autorização de atualização sobre *empréstimo* a U_j. A raiz do gráfico é o administrador de banco de dados. No gráfico de exemplo da Figura 8.11, observe que o usuário U_5 recebe autorização por U_1 e U_2; U_4 recebe autorização somente por U_1.

Um usuário possui uma autorização *se e somente* se houver um caminho da raiz do gráfico de autorização (a saber, o nó representando o administrador de banco de dados) até o nó representando o usuário.

Suponha que o administrador de banco de dados decida revogar a autorização do usuário U_1. Como U_4 tem autorização de U_1, essa autorização também deve ser revogada. Porém, U_5 recebeu autorização por U_1 e U_2. Como o administrador de banco de dados não revogou autorização update sobre *empréstimo* de U_2, U_5 retém autorização update sobre *empréstimo*. Se U_2 por fim revogar a autorização de U_5, então U_5 pede a autorização.

Um par de usuários maliciosos poderia tentar derrubar as regras para revogação de autorização concedendo autorização um ao outro, como mostra a Figura 8.12a. Se o administrador de banco de dados revogar a autorização de U_2, U_2 retém a autorização por U_3, como na Figura 8.12b. Se a autorização for revogada mais tarde de U_3, U_3 parece reter autorização por meio de U_2, como na Figura 8.12c. Porém, quando o administrador de banco de dados revogar a autorização de U_3, as arestas de U_3 para U_2 e de U_2 para U_3 não fazem mais parte de um caminho começando com o administrador de banco de dados. Exigimos que todas as arestas em um gráfico de autorização façam parte de algum caminho originando com o administrador de banco de dados. As arestas entre U_2 e U_3 são excluídas, e o gráfico de autorização resultante é como na Figura 8.13.

Concessão de privilégios em SQL

Vimos a sintaxe SQL básica para conceder e revogar privilégios na seção "Autorização" do Capítulo 4. Lembre-se de que a instrução **grant** é usada para conferir autorização. O formato básico dessa instrução é:

grant <*lista privilégios*> **on** <*nome relação ou nome view*> **to** <*lista usuários/roles*>

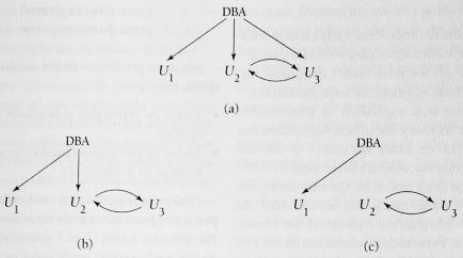


Figura 8.12 Tentativa de derrotar a revogação de autorização.

A *lista de privilégios* permite a concessão de vários privilégios em um comando. A noção de roles é abordada mais tarde, na próxima seção.

A seguinte instrução **grant** concede aos usuários U_1, U_2 e U_3 o privilégio **select** sobre a relação *conta*:

grant select on *conta* **to** U_1, U_2, U_3

O privilégio update pode ser dado sobre todos os atributos da relação ou apenas sobre alguns. Se o privilégio **update** for incluído em uma instrução **grant**, a lista de atributos sobre os quais a autorização update deve ser concedida opcionalmente aparece entre parênteses, imediatamente após a palavra-chave **update**. Se a lista de atributos for omitida, o privilégio update será concedido sobre todos os atributos da relação.

Essa instrução **grant** dá aos usuários U_1, U_2 e U_3 autorização update sobre o atributo *quantia* da relação *empréstimo*:

grant update (*quantia*) **on** *empréstimo* **to** U_1, U_2, U_3

O privilégio **insert** também pode especificar uma lista de atributos; quaisquer inserções na relação precisam especificar apenas esses atributos, e o sistema ou dá a cada um dos atributos restantes valores-padrão (se um padrão for definido para o atributo) ou os define como null.

O nome de usuário **public** refere-se a todos os usuários atuais e futuros do sistema. Assim, os privilégios concedidos a **public** são concedidos implicitamente a todos os usuários atuais e futuros.

O privilégio **references** da SQL é concedido sobre atributos específicos de uma maneira semelhante àquela para o privilégio **update**. A seguinte instrução **grant** permite que o usuário U_1 crie relações que referenciam a chave *nome_agência* da relação *agência* como uma chave estrangeira:

grant references (*nome_agência*) **on** *agência* **to** U_1

Como padrão, um usuário/role que recebe um privilégio não está autorizado a conceder esse privilégio a outro usuário/role. Se quisermos conceder um privilégio e permitir que quem o recebe passe o privilégio adiante para outros usuários, anexamos a cláusula **with grant option** ao comando **grant** apropriado. Por exemplo, se quisermos permitir que U_1 tenha o privilégio **select** sobre *agência* e permitir que U_1 conceda esse privilégio a outros, escrevemos

grant select on *agência* **to** U_1 **with grant option**

O criador de um objeto (relação/view/role) recebe todos os privilégios sobre o objeto, incluindo o privilégio de conceder privilégios a outros.

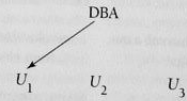


Figura 8.13 Gráfico de autorização.

Roles

Considere um banco com muitos caixas. Cada caixa precisa ter os mesmos tipos de autorizações para o mesmo conjunto de relações. Sempre que um novo caixa é designado, ele terá de receber todas essas autorizações individualmente.

Um esquema melhor seria especificar as autorizações que cada caixa precisa receber e identificar separadamente quais usuários do banco de dados são caixas. O sistema pode usar essas duas informações para determinar as autorizações de cada pessoa que é um caixa. Quando uma nova pessoa for contratada como caixa, um identificador de usuário terá de ser alocado para ela, e ela terá de ser identificada como um caixa. Permissões individuais dadas aos caixas não precisam ser especificadas novamente.

A noção de **roles** captura esse esquema. Um conjunto de roles é criado no banco de dados. As autorizações podem ser concedidas a roles, exatamente da mesma forma como são concedidas a usuários individuais. Cada usuário do banco de dados recebe um conjunto de roles (que pode ser vazio), que ele ou ela tem autorização para realizar.

Em nosso banco de dados bancário, alguns exemplos de roles poderiam ser *caixa*, *gerente_agência*, *auditor* e *administrador_sistema*.

Uma alternativa menos preferível seria criar uma userid *caixa* e permitir que cada *caixa* se conecte ao banco de dados usando a userid *caixa*. O problema com esse esquema é que não seria possível identificar exatamente qual *caixa* executou uma transação, ocasionando riscos à segurança. O uso de roles possui o benefício de exigir que os usuários se conectem ao banco de dados com sua própria userid.

Qualquer autorização que possa ser concedida a um usuário pode ser concedida a uma role. As roles são concedidas aos usuários exatamente como as autorizações. E como outras autorizações, um usuário também pode receber autorização para conceder uma role em particular a outros. Assim, os gerentes de agência podem receber autorização para conceder a role *caixa*.

As roles podem ser criadas na SQL:1999 da seguinte maneira:

create role *caixa*

As roles podem, então, receber privilégios assim como os usuários, conforme ilustramos nesta instrução:

grant select on *conta* **to** *caixa*

As roles podem ser concedidas a usuários, bem como a outras roles, como mostram as instruções a seguir.

grant *caixa* **to** john
create role *gerente*
grant *caixa* **to** *gerente*
grant *gerente* **to** mary

Assim, os privilégios de um usuário ou de uma role consistem em

- Todos os privilégios concedidos diretamente ao usuário/role
- Todos os privilégios concedidos a roles que foram concedidas ao usuário/role

Observe que pode haver uma cadeia de roles; por exemplo, a role *funcionário* pode ser concedida a todos os *caixas*. Por sua vez, a role *caixa* é concedida a todos os *gerentes*. Assim, a role *gerente* herda todos os privilégios concedidos às roles *funcionário* e *caixa*, além dos privilégios concedidos diretamente a *gerente*.

As ações executadas por uma sessão possuem todos os privilégios concedidos diretamente ao usuário, além de todos os privilégios concedidos às roles que são concedidas (direta ou indiretamente por outras roles) a esse usuário. Assim, se um usuário John tiver recebido a role *gerente*, as ações executadas pelo usuário John recebem todos os privilégios concedidos diretamente a John, além dos privilégios concedidos a *gerente*, mais os privilégios concedidos a *caixa* se a role *caixa* foi concedida à role *gerente*.

Além da noção do usuário (atual) de uma sessão, a SQL também tem uma noção da role atual associada a uma sessão. Como padrão, a role atual associada a uma sessão é nula (exceto em alguns casos especiais). A role atual associada a uma sessão pode ser definida pela execução de **set role** *nome_role*. A role especificada precisa ter sido concedida ao usuário, ou então a instrução **set role** falha.

Quando um privilégio é concedido, como padrão, ele é tratado como tendo sido concedido pelo usuário atual, ou seja, o concessor é o usuário atual. Para conceder um privilégio com o concessor definido como a role atual associada a uma sessão, podemos acrescentar a cláusula **granted by current_role** à instrução grant, desde que a role atual não seja nula. A motivação para especificar que o concessor de um privilégio é uma role especificada se tornará clara mais tarde, quando discutirmos a revogação de privilégios.

Revogação de privilégios

Para revogar uma autorização, usamos a instrução **revoke**. Ela tem um formato quase idêntico ao do **grant**:

revoke <*lista privilégios*> **on** <*nome relação ou view*>
from <*lista usuários/roles*>> [**restrict** | **cascade**]

Assim, para revogar os privilégios que concedemos anteriormente, escrevemos

revoke select on *agência* **from** U_1, U_2, U_3
revoke update (*quantia*) **on** *empréstimo* **from** U_1, U_2, U_3
revoke references (*nome_agência*) **on** *agência* **from** U_1

Como vimos na seção "Concessão de privilégios", a revogação de um privilégio de um usuário/role pode fazer com que outros usuários/roles também percam esse privilégio. Esse comportamento é chamado de *propagação de revogação*. Na maioria dos sistemas de banco de dados, a propagação é o comportamento padrão; a palavra-chave **cascade**, assim, poderá ser omitida, como fizemos nos exemplos anteriores. A instrução **revoke**, como alternativa, pode especificar **restrict**:

revoke select on *agência* **from** U_1, U_2, U_3 **restrict**

Nesse caso, o sistema retorna um erro se houver quaisquer propagações de revogação e não executa a ação de revogar. A seguinte instrução **revoke** revoga apenas a opção grant, em vez do privilégio **select** real:

revoke grant option for select on *agência* **from** U_1

A propagação de revogações é imprópria em muitas situações. Suponha que Mary tenha a role de *gerente*, concede *caixa* a John e depois a role *gerente* é revogada de Mary (talvez porque Mary sai da empresa); John continua empregado e deverá reter a role *caixa*.

Para lidar com essa situação, a SQL:1999 permite que um privilégio seja concedido por uma role em vez de um usuário. Suponha que a concessão da role *caixa* (ou outros privilégios) a John seja feita com o concessor definido como a role *gerente* (usando a cláusula **granted by current_role** que vimos anteriormente, com a role atual definida como *gerente*), em vez de o concessor ser o usuário Mary. Então, a revogação de roles/privilégios (incluindo a role *gerente*) de Mary não resultará na revogação de privilégios que tinham o concessor definido como a role *gerente*, mesmo que Mary fosse o usuário que executou o grant; assim, John reteria a role *caixa* mesmo depois de os privilégios de Mary serem revogados.

Autorização sobre views, funções e procedimentos

No Capítulo 3, apresentamos o conceito de views como um meio de oferecer a um usuário um modelo personalizado do banco de dados. Uma view pode esconder dados que um usuário não precisa ver. A capacidade das views de ocultar dados serve tanto para simplificar o uso do sistema quanto para melhorar a segurança. As views simplificam o uso do sistema, pois restringem a atenção do usuário aos dados de interesse. Embora um usuário possa ter o acesso direto negado a uma relação, ele pode ter permissão para acessar parte dessa relação por meio de uma view. Assim, uma combinação de segurança em nível relacional e segurança em nível de view limita o acesso de um usuário aos dados exatos de que ele precisa.

Em nosso exemplo de banco, considere um funcionário que precisa saber os nomes de todos os clientes que têm um empréstimo em cada agência. Esse funcionário não está autorizado a ver informações com relação a empréstimos específicos que o cliente possa ter. Assim, o funcionário precisa ter acesso negado à relação *empréstimo*. Contudo, se precisar ter acesso às informações necessárias, o funcionário precisa receber acesso à view *cliente_empréstimo*, que consiste apenas nos nomes dos clientes e nas agências em que eles têm um empréstimo. Essa view pode ser definida em SQL da seguinte forma:

create view *cliente_empréstimo* **as**
(**select** *nome_agência*, *nome_cliente*
from *tomador*, *empréstimo*
where *tomador.número_empréstimo* = *empréstimo.número_empréstimo*)

Suponha que o funcionário emita a seguinte consulta SQL:

select *
from *cliente_empréstimo*

Logicamente, o funcionário está autorizado a ver o resultado dessa consulta. Porém, quando o processador de consulta a traduzir para uma consulta sobre as relações reais no banco de dados, produzirá uma consulta sobre *tomador* e *empréstimo*. Assim, o sistema precisa verificar a autorização sobre a consulta do funcionário antes de iniciar o processamento da consulta.

Um usuário que cria uma view não necessariamente recebe todos os privilégios sobre essa view. Ele recebe apenas os privilégios que não oferecem autorização adicional além daquelas que ele já tinha. Por exemplo, um usuário não pode receber autorização de **update** sobre uma view sem ter autorização de **update** sobre as relações usadas para definir a view. Se um usuário criar uma view em que nenhuma autorização possa ser concedida, o sistema negará a solicitação de criação da view. Em nosso exemplo de view *cliente_empréstimo*, o criador da view precisará ter autorização de **read** sobre as relações *tomador* e *empréstimo*.

O privilégio de **execute** pode ser concedido em uma função ou procedimento, permitindo que um usuário execute a função/procedimento. Como padrão, assim como as views, as funções e os procedimentos têm todos os privilégios que

o criador da função ou procedimento tinham. Com efeito, a função ou procedimento é executado como se fosse invocado pelo usuário que criou a função. O usuário atual da sessão é definido como o criador da função ou procedimento enquanto estiver em execução.

Embora esse comportamento seja apropriado em muitas situações, nem sempre é apropriado. Na SQL:2003, se a definição de função tiver uma cláusula extra **sql security invoker**, então ela será executada sob os privilégios do usuário que invoca a função, no lugar dos privilégios do definidor da função. Isso permite a criação de bibliotecas de funções que podem ser executadas sob a mesma autorização de quem chamou.

Limitações da autorização em SQL

Os padrões SQL atuais para autorização possuem alguns problemas. Por exemplo, suponha que você queira que todos os alunos possam ver suas próprias notas, mas não as notas de qualquer outro. A autorização precisa ser no nível de tuplas individuais, o que não é possível nos padrões de autorização da SQL.

Além do mais, com o crescimento na Web, os acessos ao banco de dados vêm principalmente de servidores de aplicação Web. Os usuários finais podem não ter identificadores de usuário individuais no banco de dados, e na realidade pode haver apenas um único identificador de usuário no banco de dados correspondente a todos os usuários de um servidor de aplicações.

Portanto, a tarefa da autorização cai sobre o servidor de aplicações; o esquema de autorização inteiro da SQL é desprezado. O benefício é que as autorizações detalhadas, como aqueles para tuplas individuais, podem ser implementadas pela aplicação. Os problemas são estes:

- O código para verificar a autorização se mistura com o restante do código da aplicação.
- Implementar a autorização por meio do código da aplicação, em vez de especificá-la de forma declarativa na SQL, torna difícil garantir a ausência de buracos. Por causa de um descuido, um dos programas de aplicação pode não verificar a autorização, permitindo que usuários não autorizados acessem dados confidenciais. Verificar se todos os programas de aplicação fazem todas as verificações de autorização exigidas envolve a leitura de todo o código do servidor de aplicações, uma tarefa formidável em um sistema grande.

Alguns sistemas de banco de dados oferecem mecanismos para autorização detalhada. Por exemplo, o **Virtual Private Database (VPD)** da Oracle permite que um administrador do sistema associe uma função a uma relação; a função retorna um predicado que precisa ser acrescentado a qualquer consulta que usa a relação (diferentes funções podem ser definidas para relações que estão sendo atualizadas). Por exemplo, a função para a relação *conta* poderia retornar um predicado como

número_conta **in**
(**select** *número_conta*
from *depositante*
where *depositante.name* = *syscontext.user_id*( ))

Esse predicado seria acrescentado à cláusula **where** de cada consulta que usasse a relação *conta*. Como resultado (supondo que o nome do depositante deva combinar com a user_id do banco de dados), cada usuário do banco de dados só pode ver as tuplas correspondentes às contas que ela possui. Assim, o VPD oferece autorização no nível de linhas específicas de uma relação e, portanto, é considerado como mecanismo de *autorização em nível de linha*.

Você precisa saber que a inclusão do predicado pode mudar bastante o significado de uma consulta. Por exemplo, se um usuário escrevesse uma consulta para encontrar o saldo médio da conta, acabaria obtendo a média dos saldos de suas próprias contas.

Para lidar com aplicações Web em que a aplicação se conecta ao banco de dados usando um único identificador de usuário, o Oracle também permite que as aplicações definam a identificação de usuário em uma conexão. Veja, nas notas bibliográficas, dicas para obter mais informações sobre Oracle VPD.

As várias provisões que um sistema de banco de dados pode fazer para autorização podem não oferecer proteção suficiente para dados altamente sensíveis. Nesses casos, os dados podem ser armazenados em formato criptografado. A criptografia é descrita com mais detalhes na seção "Técnicas de criptografia".

Trilhas de auditoria

Muitas aplicações de banco de dados seguras exigem que uma **trilha de auditoria** seja mantida. Uma trilha de auditoria é um registro de todas as mudanças (inserts/deletes/updates) ao banco de dados, junto com informações como qual usuário realizou a mudança e quando ela foi realizada.

A trilha de auditoria ajuda a segurança de várias maneiras. Por exemplo, se for descoberto que o saldo de uma conta está incorreto, o banco pode querer rastrear todas as atualizações realizadas na conta para descobrir as atualizações incorretas (ou fraudulentas), além das pessoas que executaram as atualizações. O banco poderia, então, usar a trilha de auditoria para rastrear todas as atualizações realizadas

por essas pessoas, a fim de encontrar outras atualizações incorretas ou fraudulentas.

É possível criar uma trilha de auditoria definindo triggers apropriados sobre atualizações de relação (usando variáveis definidas pelo sistema que identificam o nome de usuário e hora). Porém, muitos sistemas de banco de dados oferecem mecanismos embutidos para criar trilhas de auditoria, que são muito mais convenientes de usar. Os detalhes de como criar trilhas de auditoria variam entre os sistemas de banco de dados, e você deverá consultar os manuais do sistema de banco de dados para obter os detalhes.

Segurança da aplicação

A segurança dos dados da aplicação precisa lidar com várias ameaças de segurança e questões além daquelas tratadas pela autorização da SQL. Por exemplo, os dados precisam ser protegidos enquanto estão sendo transmitidos; eles podem ter de ser protegidos contra intrusos capazes de burlar a segurança do sistema operacional e podem ter restrições de privacidade complexas, que vão além daquilo que um banco de dados pode impor. Consideramos esses e outros aspectos relacionados nesta seção.

Técnicas de criptografia

Existe uma grande variedade de técnicas para criptografia de dados. As técnicas de criptografia simples podem não oferecer segurança adequada, pois pode ser fácil para um usuário não autorizado quebrar o código. Como exemplo de uma técnica de criptografia fraca, considere a substituição de cada caractere pelo próximo caractere no alfabeto. Assim,

Perryridge

torna-se

Qfsszsjehf

Se um usuário não autorizado vir apenas "Qfsszsjehf", ele provavelmente terá informações insuficientes para quebrar o código. Porém, se o intruso encontrar uma série de nomes de agência criptografados, ele poderá usar dados estatísticos com relação à freqüência relativa dos caracteres para descobrir qual substituição está sendo feita (por exemplo, *E* é a letra mais comum no texto em inglês, seguida por *T*, *A*, *O*, *N*, *I* e assim por diante).

Uma boa técnica de criptografia tem as seguintes propriedades:

- Ela é relativamente simples para usuários autorizados codificarem e decodificarem dados.
- Ela não depende do segredo do algoritmo, mas de um parâmetro do algoritmo, chamado *chave de criptografia*.
- Sua chave de criptografia é extremamente difícil para um intruso determinar.

Uma técnica, a *Data Encryption Standard* (DES), lançada em 1977, realiza uma substituição de caracteres e uma modificação de sua ordem com base em uma chave de criptografia. Para que esse esquema funcione, os usuários autorizados precisam receber a chave de criptografia por meio de um mecanismo seguro. Esse requisito é um problema sério, pois o esquema não é mais seguro do que a segurança do mecanismo pelo qual a chave de criptografia é transmitida. O padrão DES foi reafirmado em 1983, 1987 e novamente em 1993. Porém, o problema no DES foi reconhecido em 1993 e chegou a um ponto em que um novo padrão, a ser chamado de **Advanced Encryption Standard** (AES), precisava ser selecionado. Em 2000, o **algoritmo Rijndael** (que tem o nome dos inventores V. Rijmen e J. Daemen), foi selecionado para ser o AES. O algoritmo Rijndael foi escolhido por seu nível de segurança muito mais forte e sua relativa facilidade de implementação nos sistemas de computador atuais, além de dispositivos como smart cards (cartões inteligentes). Assim como o padrão DES, Rijndael é um algoritmo de chave compartilhada (ou chave simétrica), em que os usuários autorizados compartilham uma chave.

A **criptografia de chave pública** é um esquema alternativo que evita alguns dos problemas que encaramos com o DES. Ele é baseado em duas chaves: uma *chave pública* e uma *chave privada*. Cada usuário U_i possui uma chave pública E_i e uma chave privada D_i. Todas as chaves públicas são publicadas: elas podem ser vistas por qualquer um. Cada chave privada é conhecida apenas pelo único usuário a quem a chave pertence. Se o usuário U_1 quiser armazenar dados criptografados, U_1 os criptografa usando a chave pública E_1. A descriptografia requer a chave privada D_1.

Como a chave de criptografia para cada usuário é pública, é possível trocar informações com segurança por meio desse esquema. Se o usuário U_1 quiser compartilhar dados com U_2, U_1 os criptografa usando E_2, a chave pública de U_2. Como somente o usuário U_2 sabe como decodificar os dados, as informações são transferidas com segurança.

Para a criptografia de chave pública funcionar, é preciso haver um esquema que pode se tornar público sem facilitar que as pessoas descubram o esquema para a decodificação. Em outras palavras, precisa ser difícil deduzir a chave privada, dada a chave pública. Tal esquema existe e é baseado nestas condições:

- Existe um algoritmo eficiente para testar se um número é primo ou não.
- Não existe um algoritmo eficiente conhecido por encontrar os fatores primos de um número.

Para os propósitos deste esquema, os dados são tratados como uma coleção de inteiros. Criamos uma chave pública calculando o produto de dois números primos grandes: P_1 e P_2. A chave privada consiste no par (P_1, P_2). O algoritmo de descriptografia não pode ser usado com sucesso se apenas o produto $P_1 P_2$ for conhecido; ele precisa dos valores individuais P_1 e P_2. Como tudo o que é publicado é o produto $P_1 P_2$, um usuário não autorizado precisaria ser capaz de fatorar $P_1 P_2$ para roubar dados. Escolhendo P_1 e P_2 para ser suficientemente grande (mais de 100 dígitos), podemos tornar o custo da fatoração de $P_1 P_2$ proibitivamente alto (na ordem de anos de tempo de cálculo, nos computadores mais velozes).

Os detalhes da criptografia de chave pública e a justificativa matemática das propriedades dessa técnica podem ser encontrados nas notas bibliográficas.

Embora a criptografia de chave pública por esse esquema seja segura, ela também é computacionalmente dispendiosa. Um esquema híbrido usado para a comunicação segura é o seguinte: as chaves DES são trocadas por meio de um esquema de criptografia de chave pública, e a criptografia DES é usada sobre os dados transmitidos mais tarde.

Suporte para criptografia nos bancos de dados

Muitos sistemas de arquivo e sistemas de banco de dados hoje admitem criptografia de dados. Essa criptografia protege os dados de alguém que é capaz de acessá-los, mas não é capaz de acessar a chave de descriptografia. No caso da criptografia do sistema de arquivos, os dados a serem codificados normalmente são grandes arquivos e diretórios contendo informações sobre arquivos.

No contexto dos bancos de dados, a criptografia pode ser feita em vários níveis diferentes. No nível mais baixo, os blocos de disco contendo dados de banco de dados podem ser criptografados, usando uma chave disponível ao software do sistema de banco de dados. Quando um bloco é recuperado do disco, ele primeiro é decodificado e depois usado pelo modo normal. Essa criptografia em nível de bloco de disco protege contra invasores que podem acessar o conteúdo do disco mas não têm acesso à chave de criptografia. Ela também tem a vantagem de exigir relativamente pouco tempo e espaço adicionais. Por exemplo, se os dados no banco de dados de um computador laptop tiverem de ser protegidos contra roubo do próprio computador, essa criptografia poderá ser usada. A chave de descriptografia teria de ser fornecida pelo usuário sempre que o software de banco de dados fosse reiniciado. De modo semelhante, alguém que tem acesso a fitas de backup não poderia acessar os dados contidos nos backups sem conhecer a chave de descriptografia.

Em um sistema de banco de dados compartilhado, a criptografia de bloco de disco não pode ser usada para proteger dados contra outros usuários privilegiados, como administradores de banco de dados, que podem emitir consultas ao banco de dados. Para proteger os dados contra esse tipo de acesso, a criptografia precisa ser feita *antes* que os dados cheguem ao banco de dados. A aplicação precisa codificar os dados antes de enviá-los ao banco de dados. Vários sistemas de banco de dados oferecem APIs para criptografia que oferecem tal suporte para colunas especificadas. Uma única chave pode ser usada para todas as colunas criptografadas e para todas as linhas para determinada coluna. O uso de uma chave diferente para cada linha não é viável, pois tornaria a tarefa de gerenciamento de chave muito difícil.

O armazenamento seguro de chaves de descriptografia é outro problema relacionado. Se elas forem armazenadas como um arquivo no sistema operacional, alguém capaz de burlar a segurança do sistema operacional poderia ter acesso às chaves. Alguns sistemas operacionais oferecem *armazenamento seguro*; ou seja, eles só permitem que a aplicação que armazenou a chave a recupere. A própria aplicação pode ser identificada por um valor de hash em seu executável, de modo que os atacantes que substituem a aplicação por uma cópia modificada não podem ter acesso à chave.

A criptografia de pequenos valores, como identificadores ou nomes, é complicada pela possibilidade de *ataques de dicionário*, principalmente se a chave de criptografia estiver disponível publicamente. Por exemplo, se os campos de data de nascimento forem criptografados, um atacante tentando descriptografar um valor criptografado em particular c poderia tentar criptografar cada data de nascimento possível até encontrar uma cujo valor criptografado combina com c. Esses ataques podem ser frustrados pelo acréscimo de bits aleatórios extras ao final do valor antes da criptografia (com sua remoção após a descriptografia). Esses bits extras (às vezes chamados de *bits de sal*) oferecem boa proteção contra ataques de dicionário.

Autenticação

A autenticação refere-se à tarefa de verificar a identidade de uma pessoa/software conectando-se a um banco de dados. A forma mais simples de autenticação consiste em uma senha secreta que precisa ser apresentada quando uma conexão for aberta com um banco de dados.

A autenticação baseada em senha é bastante usada pelos sistemas operacionais e também pelos bancos de dados. Porém, o uso de senhas possui algumas desvantagens, especialmente por uma rede. Se um bisbilhoteiro for capaz de "farejar" os dados sendo enviados pela rede, ele pode ser capaz de encontrar a senha enquanto estiver sendo enviada

pela rede. Quando o bisbilhoteiro tiver um nome de usuário e senha, ele poderá se conectar com o banco de dados, fingindo ser o usuário legítimo.

Sistemas desafio-resposta

Um esquema mais seguro envolve um sistema de **desafio-resposta**. O sistema de banco de dados envia uma string de desafio para o usuário. O usuário codifica a string de desafio usando uma senha secreta como chave de criptografia e depois retorna o resultado. O sistema de banco de dados pode verificar a autenticidade do usuário decodificando a string com a mesma senha secreta e verificando o resultado com a string de desafio original. Esse esquema garante que nenhuma senha atravessará a rede.

Os sistemas de chave pública podem ser usados para criptografia nos sistemas de desafio-resposta. O sistema de banco de dados criptografa uma string de desafio usando a chave pública do usuário e a envia ao usuário. O usuário descriptografa a string usando sua chave privada e retorna o resultado ao sistema de banco de dados. O sistema de banco de dados, então, verifica a resposta. Esse esquema possui o benefício adicional de não armazenar a senha secreta no banco de dados, onde potencialmente poderia ser visto pelos administradores do sistema.

O armazenamento da chave privada de um usuário em um computador (até mesmo um computador pessoal) tem o risco de que, se o computador for comprometido, a chave pode ser revelada a um atacante que poderá então fingir ser um usuário. Os **smart cards** oferecem uma solução para esse problema. Em um smart card, a chave pode ser armazenada em um chip embutido; o sistema operacional do smart card garante que a chave nunca poderá ser lida, mas permite que os dados sejam enviados ao cartão para criptografia ou descriptografia, usando a chave privada.[3]

Assinaturas digitais

Outra aplicação interessante da criptografia de chave pública está nas assinaturas digitais, para verificar a autenticidade dos dados. As assinaturas digitais desempenham o papel eletrônico das assinaturas físicas nos documentos. A chave privada é usada para assinar dados, e os dados assinados podem se tornar públicos. Qualquer um pode verificá-los pela chave pública, mas ninguém poderia ter gerado os dados assinados sem ter a chave privada. Assim, podemos **autenticar** os dados; ou seja, podemos verificar se eles foram realmente criados pela pessoa que afirma tê-los criado.

Além do mais, as assinaturas digitais também servem para garantir o **não repúdio**. Ou seja, caso a pessoa que criou os dados mais tarde reclame que não os criou (o equivalente eletrônico de declarar não ter assinado o cheque), podemos provar que essa pessoa criou os dados (a menos que sua chave privada fosse passada para outros).

Certificados digitais

A autenticação em geral é um processo duplo, em que cada um de um par de entidades interagindo se autentica com o outro. Essa autenticação em nível de par é necessária mesmo quando um cliente entra em contato com um site, para impedir que um site malicioso finja ser um site válido. Esse disfarce poderia ser feito, por exemplo, se os roteadores da rede fossem comprometidos, e os dados redirecionados para o site malicioso.

Para um usuário garantir que está interagindo com um site autêntico, ele precisa ter a chave pública do site. Isso nos remete ao problema de como o usuário pode obter a chave pública – se ela for armazenada no site, o site malicioso poderia fornecer uma chave diferente, e o usuário não teria como verificar se a chave pública fornecida é autêntica. A autenticação pode ser tratada por um sistema de **certificados digitais**, pelos quais as chaves públicas são assinadas por uma agência de certificação, cuja chave pública é bem conhecida. Por exemplo, as chaves públicas das autoridades de certificação raiz são armazenadas em navegadores Web padrão. Um certificado emitido por elas pode ser verificado pelo uso das chaves públicas armazenadas.

Um sistema de dois níveis colocaria um peso excessivo de criação de certificados sobre as autoridades de certificação raiz, de modo que é utilizado um sistema multinível em seu lugar, com uma ou mais autoridades de certificação raiz e uma árvore de autoridades de certificação abaixo de cada raiz. Cada autoridade (diferente das autoridades raiz) possui um certificado digital emitido por seu pai.

Um certificado digital emitido por uma autoridade de certificação A consiste em uma chave pública K_A e um texto criptografado E, que pode ser decodificado por meio da chave pública K_A. O texto criptografado contém o nome da parte a quem o certificado foi emitido e sua chave pública K_c. Caso a autoridade de certificação A não seja a autoridade de certificação raiz, o texto criptografado também possui o certificado digital emitido para A por sua autoridade de certificação pai; esse certificado autentica a própria chave K_A. (Esse certificado, por sua vez, pode conter um certificado de outra autoridade pai, e assim por diante.)

Para verificar um certificado, o texto criptografado E é decodificado com o uso da chave pública, e se A não for uma autoridade raiz, a chave pública K_A é verificada recursivamente pelo uso do certificado digital contido dentro de E; a

3. Os smart cards também oferecem outra funcionalidade, como a capacidade de armazenar dinheiro e fazer pagamentos digitalmente, o que não é relevante em nosso contexto.

recursão termina quando um certificado emitido pela autoridade raiz é alcançado. A verificação do certificado estabelece a cadeia pela qual determinado site foi autenticado, e oferece o nome e a chave pública autenticada para o site.

Os certificados digitais são muito utilizados para autenticar sites aos usuários, de modo a impedir que sites maliciosos sejam mascarados como outros sites. No protocolo HTTPS (a versão segura do protocolo http), o site oferece seu certificado digital ao navegador, que então o exibe ao usuário. Se o usuário aceitar o certificado, o navegador emitirá a chave pública oferecida aos dados codificados. Um site malicioso terá acesso ao certificado, mas não à chave privada, e por isso não poderá decodificar os dados enviados pelo navegador. Somente o site autêntico, que possui a chave privada correspondente, pode decodificar os dados enviados pelo navegador. Observamos que os custos de criptografia e descriptografia de chave pública/privada são muito maiores do que os custos de criptografia/descriptografia usando chaves privadas simétricas. Para reduzir os custos de criptografia, o HTTPS realmente cria uma chave simétrica de uma vez após a autenticação, e a usa para codificar dados pelo restante da sessão.

Os certificados digitais também podem ser usados para autenticar usuários. O usuário precisa submeter um certificado digital contendo sua chave pública a um site, que verifica se o certificado foi assinado por uma autoridade de confiança. A chave pública do usuário pode, então, ser usada em um sistema de desafio-resposta para garantir que o usuário possui a chave privada correspondente, autenticando assim o usuário.

Autenticação central

Quando os usuários acessam múltiplos sites, normalmente é incômodo ter de se autenticar em cada site separadamente, normalmente com diferentes senhas em cada um. Existem sistemas que permitem que o usuário se autentique em um serviço de autenticação central, e outros sites podem autenticar o usuário por meio desse site; a mesma senha pode, então, ser usada para acessar vários sites.

Um sistema de **logon único** permite ainda que o usuário seja autenticado uma vez (normalmente, inserindo uma senha) e que várias aplicações possam verificar a identidade do usuário por meio do serviço de autenticação central sem exigir nova autenticação. Esses mecanismos de logon único há muito tempo têm sido usados nos sistemas operacionais distribuídos, como Kerberos, e agora existem implementações para aplicações Web. Veja mais informações nas notas bibliográficas.

Além de autenticar usuários, um serviço de autenticação central pode oferecer outros serviços, como informações sobre o usuário (por exemplo, nome, endereço de e-mail e informações de endereço) à aplicação. Isso elimina a necessidade de entrar com essas informações separadamente em cada aplicação. Os sistemas de diretório como LDAP e Active Directories, e sistemas de autenticação como o serviço Passport da Microsoft, oferecem mecanismos para autenticar usuários e também para oferecer informações sobre o usuário.

Protegendo aplicações

Existem muitas maneiras de comprometer a segurança de uma aplicação, mesmo que o próprio sistema de banco de dados seja seguro. Esboçamos algumas brechas de segurança em potencial e como proteger contra elas.

Em ataques de **injeção de SQL**, o atacante consegue fazer com que uma aplicação execute uma consulta SQL criada por ele. Alguns ataques funcionam da seguinte maneira. Considere o texto de código-fonte mostrado na Figura 8.2. Suponha que o servlet correspondente, mostrado na Figura 8.6, crie uma string de consulta usando a seguinte expressão Java:

```
"select saldo from conta where número_conta" =' + número + "'"
```

onde `número` é uma variável contendo a string inserida pelo usuário. Um atacante malicioso usando o formulário Web pode digitar uma string como "';<alguma instrução SQL>;'", onde <alguma instrução> indica qualquer instrução SQL que o atacante deseja, no lugar de um número de conta válido. O servlet, então, criaria e submeteria a string a seguir.

```
select saldo from conta where número_conta = '';
    <alguma instrução SQL>;
```

O apóstrofo inserido pelo atacante fecha a string, o ponto-e-vírgula seguinte termina a consulta e o texto seguinte inserido pelo atacante é interpretado como uma consulta SQL. Assim, o usuário malicioso conseguiu inserir uma instrução SQL qualquer, que é executada pela aplicação. A instrução pode causar danos significativos, pois pode contornar todas as medidas de segurança implementadas no código da aplicação.

Para evitar tais ataques, é melhor usar **instruções preparadas** para executar consultas SQL. Ao definir um parâmetro de uma consulta preparada, o JDBC acrescenta automaticamente caracteres de escape para que os apóstrofos fornecidos pelo usuário não sejam mais capazes de terminar a string. De modo equivalente, uma função que acrescenta esses caracteres de escape poderia ser aplicada às strings de entrada antes de serem concatenadas à consulta SQL, em vez de usar instruções preparadas.

Outro problema que os desenvolvedores de aplicação precisam lidar é o armazenamento de senhas em texto

limpo no código da aplicação. Por exemplo, programas como scripts JSP normalmente contêm senhas em texto limpo. Se esses scripts forem armazenados em um diretório acessível por um servidor Web, um usuário externo pode ser capaz de acessar o código-fonte do script e obter acesso à senha para a conta do banco de dados usada pela aplicação. Para evitar esses problemas, muitos servidores de aplicação oferecem mecanismos para armazenar senhas em forma criptografada, que o servidor decodifica antes de passá-las ao banco de dados. Esse recurso remove a necessidade de armazenar senhas como texto limpo nos programas de aplicação.

Como outra medida contra senhas de banco de dados comprometidas, muitos sistemas de banco de dados permitem que o acesso ao banco de dados seja restrito a determinado conjunto de endereços da Internet. As tentativas de conexão com o banco de dados a partir de outros endereços da Internet são rejeitadas.

Privacidade

Em um mundo onde uma quantidade cada vez maior de dados pessoais está disponível on-line, as pessoas cada vez mais se preocupam com a privacidade de seus dados. Por exemplo, a maioria das pessoas desejaria que seus dados médicos pessoais fossem mantidos privados e não revelados publicamente. Porém, os dados médicos precisam se tornar disponíveis aos médicos e aos técnicos de emergência que tratam do paciente. Muitos países têm leis de privacidade desses dados, que definem quando e a quem os dados podem ser revelados. A violação da lei de privacidade pode resultar em penalidades criminosas em alguns países. As aplicações que acessam esses dados privados precisam ser montadas com cuidado, lembrando das leis de privacidade.

Por outro lado, dados privados agregados podem desempenhar um papel importante em muitas tarefas, como a detecção de efeitos colaterais das drogas, ou na detecção de espalhamento de epidemias. Como tornar esses dados disponíveis aos pesquisadores que executam tais tarefas, sem comprometer a privacidade dos indivíduos, é um problema importante no mundo real. Como exemplo, suponha que um hospital esconda o nome do paciente, mas ofereça a um pesquisador a data de nascimento e o código postal (CEP) do paciente (ambos úteis para o pesquisador). Apenas essas duas informações podem ser usadas para identificar com exclusividade o paciente em muitos casos (usando informações de um banco de dados externo), comprometendo sua privacidade. Nessa situação específica, uma solução seria dar o ano de nascimento, mas não a data de nascimento, junto com o código postal, o que pode bastar para o pesquisador. Isso não ofereceria informações suficientes para identificar a maioria dos indivíduos de forma exclusiva.[4]

Como outro exemplo, os sites normalmente coletam dados pessoais como informações de endereço, telefone, e-mail e cartão de crédito. Essas informações podem ser necessárias para executar uma transação como a compra de um item de uma loja. Porém, o cliente pode não querer que a informação se torne disponível para outras organizações, ou pode querer que parte da informação (como números de cartão de crédito) seja apagada após algum período de tempo, como um meio de impedir que ela caia em mãos não autorizadas, se houver brecha na segurança. Muitos sites permitem que os clientes especifiquem suas preferências de privacidade, e precisam garantir que essas preferências sejam respeitadas.

Resumo

- A maioria dos usuários interage com bancos de dados por meio de formulários e interfaces gráficas com o usuário, e existem várias ferramentas para simplificar a construção dessas interfaces. Os geradores de relatórios são ferramentas que ajudam a criar relatórios legíveis às pessoas a partir do conteúdo do banco de dados.
- O navegador Web se tornou a interface com o usuário mais usada para bancos de dados. HTML oferece a capacidade de definir interfaces que combinam hyperlinks com facilidades de formulário. Os navegadores Web se comunicam com os servidores Web pelo protocolo HTTP. Os servidores Web podem passar solicitações adiante para os programas de aplicação, e retornar os resultados ao navegador.
- Existem várias linguagens de scripting no cliente – JavaScript é a mais utilizada – que oferecem interfaces com o usuário mais ricas na extremidade do navegador.
- Os servidores Web executam programas de aplicação para implementar a funcionalidade desejada. Os servlets são um mecanismo bastante utilizado para escrever programas de aplicação executados como parte do processo do servidor Web, a fim de reduzir as sobrecargas. Há também muitas linguagens de scripting no servidor que são interpretadas pelo servidor Web e oferecem funcionalidade de programa de aplicação como parte do servidor Web.
- Triggers definem ações a serem executadas automaticamente quando certos eventos ocorrem e condições correspondentes são satisfeitas. Triggers possuem muitos usos, como a implementação de regras de negócios, log-

4. Para pessoas extremamente idosas, que são relativamente raras, até mesmo o ano de nascimento mais o código postal podem ser suficientes para identificar o indivíduo de forma exclusiva, de modo que um intervalo de valores, como 80 anos ou mais, pode ser fornecido no lugar da idade real para pessoas com mais de 80 anos.

ging de auditoria e até mesmo execução de ações fora do sistema de banco de dados. Embora os triggers tenham sido incluídos bem tarde no padrão SQL como parte da SQL:1999, a maioria dos sistemas de banco de dados há muito tempo já os implementa.

- Um usuário que recebeu alguma forma de autoridade pode ter permissão para passar essa autoridade a outros usuários. No entanto, é preciso ter cuidado em como a autorização pode ser passada entre os usuários se tivermos de garantir que ela pode ser revogada em algum momento no futuro.
- As roles ajudam a atribuir um conjunto de privilégios a um usuário de acordo com a função que ele desempenha na organização.
- Os mecanismos de autorização SQL são genéricos e de valor limitado para aplicações que lidam com grandes quantidades de usuários. Foram desenvolvidas extensões para oferecer controle de acesso em nível de linha e lidar com grandes quantidades de usuários de aplicação, mas elas ainda não são padronizadas.
- A criptografia desempenha um papel-chave na proteção de informações e na autenticação de usuários e sites. Os sistemas de desafio-resposta normalmente são usados para autenticar usuários. Os certificados digitais desempenham um papel-chave na autenticação de sites.
- Os desenvolvedores de aplicação precisam prestar muita atenção à segurança, para evitar ataques de injeção de SQL e outros ataques por usuários maliciosos.
- A proteção da privacidade dos dados é uma tarefa importante para aplicações de banco de dados. Muitos países têm requisitos legais sobre a manutenção de privacidade de certos tipos de dados, como os dados médicos.

Termos de revisão

- Formulários
- Interfaces gráficas com o usuário
- Geradores de relatório
- Interfaces Web com bancos de dados
- HyperText Markup Language (HTML)
- Hyperlinks
- Uniform Resource Locator (URL)
- Scripting no servidor
- JavaScript
- Document Object Model (DOM)
- Applets
- Linguagem de scripting no cliente
- Servidores Web
- Sessão
- HyperText Transfer Protocol (HTTP)
- Common Gateway Interface (CGI)
- Sem conexão
- Cookie
- Servlets
- Sessões de servlet
- JSP
- Scripting no servidor
- Pooling de conexão
- ASP.NET
- Trigger
- Modelo evento-condição-ação
- Triggers before e after
- Variáveis e tabelas de transição
- Autorização
- Privilégios
- Privilégio para conceder privilégios
- Opção de grant
- Roles
- Revogação de privilégios
- Autorização sobre views
- Autorização execute
- Privilégios de invocador
- Autorização em nível de linha
- Trilhas de auditoria
- Criptografia
- Criptografia de chave pública
- Autenticação
- Desafio-resposta
- Assinaturas digitais
- Certificados digitais
- Autenticação central
- Logon único
- Injeção de SQL
- Privacidade

Exercícios práticos

8.1 Qual é o principal motivo para os servlets terem melhor desempenho do que os programas que usam a Common Gateway Interface (CGI), embora os programas Java geralmente sejam mais lentos do que programas em C ou C++?

8.2 Liste alguns benefícios e desvantagens dos protocolos sem conexão em relação aos protocolos que mantêm conexões.

8.3 Liste três maneiras como o caching pode ser usado para agilizar o desempenho do servidor Web.

8.4 Considere uma view *agência_cli* definida da seguinte forma:

```
create view agência_cli as
select nome_agência, nome_cliente
from depositante, conta
where depositante.número_conta = conta.número_conta
```

Suponha que a view seja *materializada*, isto é, a view é calculada e armazenada. Escreva triggers para *manter* a view, ou seja, para mantê-la atualizada nas inserções e exclusões de *depositante* ou *conta*. Não se preocupe com as atualizações.

8.5 Escreva um trigger SQL para executar a ação a seguir. Na exclusão de uma conta, para cada proprietário de conta, verifique se o proprietário tem quaisquer contas restantes e, se não tiver, exclua-o da relação *depositante*.

8.6 Suponha que alguém personifique uma empresa e receba um certificado de uma autoridade de emissão de certificado. Qual é o efeito sobre coisas (como ordens de compra ou programas) certificadas pela empresa personificada, e sobre coisas certificadas por outras empresas?

8.7 Talvez os itens de dados mais importantes em qualquer sistema de banco de dados sejam as senhas que controlam o acesso ao banco de dados. Sugira um esquema para o armazenamento seguro de senhas. Cuide para que seu esquema permita que o sistema teste as senhas fornecidas pelos usuários que estão tentando efetuar login no sistema.

Exercícios

8.8 Escreva um servlet e o código HTML associado para a seguinte aplicação muito simples: um usuário tem permissão para submeter um formulário contendo um valor, digamos *n*, e deve receber uma resposta contendo *n* símbolos "*".

8.9 Escreva um servlet e o código HTML associado para a seguinte aplicação simples: um usuário tem permissão para submeter um formulário contendo um número, digamos *n*, e deve receber uma resposta dizendo quantas vezes o valor *n* foi submetido anteriormente. O número de vezes que cada valor foi submetido anteriormente deve ser armazenado em um banco de dados.

8.10 Escreva um servlet que autentica um usuário (com base nos nomes de usuário e senhas armazenadas em uma relação de banco de dados) e define uma variável de sessão chamada *userid* após a autenticação.

8.11 O que é um ataque de injeção de SQL? Explique como ele funciona e que precauções precisam ser tomadas para impedir ataques de injeção de SQL.

8.12 Escreva o pseudocódigo para gerenciar um pool de conexão. Seu pseudocódigo precisa incluir uma função para criar um pool (oferecendo uma string de conexão de banco de dados, nome de usuário de banco de dados e senha como parâmetros), uma função para solicitar uma conexão do pool, uma conexão para liberar uma conexão com o pool e uma função para fechar o pool de conexão.

8.13 Suponha que existam duas relações *r* e *s*, de modo que a chave estrangeira *B* de *r* referencia a chave primária *A* de *s*. Descreva como o mecanismo de trigger pode ser usado para implementar a opção **on delete cascade**, quando uma tupla for excluída de *s*.

8.14 A execução de um trigger pode fazer com que outra ação seja disparada. A maioria dos sistemas de banco de dados coloca um limite sobre a profundidade que o aninhamento pode ter. Explique por que eles precisam colocar esse limite.

8.15 Explique por que, quando uma gerente (digamos, Mary) concede uma autorização, o grant deve ser feito pela role de gerente, e não pelo usuário Mary.

8.16 Suponha que o usuário *A*, que tem todas as autorizações sobre uma relação *r*, conceda select sobre a relação *r* a **public** com opção de grant. Suponha que o usuário *B*, depois, conceda select sobre *r* a *A*. Isso causaria um ciclo no gráfico de autorização? Explique por quê.

8.17 Faça uma lista de aspectos de segurança para um banco. Para cada item na sua lista, indique se esse aspecto se relaciona a segurança física, segurança humana, segurança do sistema operacional ou segurança do banco de dados.

8.18 Os sistemas de banco de dados que armazenam cada relação em um arquivo separado do sistema operacional podem usar o esquema de segurança e autorização do sistema operacional, em vez de definir um esquema especial por si sós. Discuta uma vantagem e uma desvantagem dessa técnica.

8.19 O mecanismo VPD do Oracle implementa a segurança em nível de linha acrescentando predicados à cláusula where de cada consulta. Dê um exemplo de um predicado que poderia ser usado para implementar a segurança em nível de linha e três consultas com as seguintes propriedades:

a. Para a primeira consulta, a consulta com o predicado adicionado dá o mesmo resultado da consulta original.

b. Para a segunda consulta, aquela com os predicados adicionais dá um resultado que é sempre um subconjunto do resultado de consulta original.

c. Para a terceira consulta, a consulta com o predicado adicionado dá respostas incorretas.

8.20 Quais são duas vantagens de criptografar dados armazenados no banco de dados?

8.21 Suponha que você queira criar uma trilha de auditoria das mudanças feitas à relação *conta*.

a. Defina triggers para criar uma trilha de auditoria, registrando a informação em uma relação chamada, por exemplo, *trilha_conta*. A informação registrada deverá incluir a user-id (considere que uma função *user_id*( ) fornece essa informação) e um time-stamp , além dos valores antigo e novo. Você também precisa oferecer o esquema da relação *trilha_conta*.

b. Essa implementação pode garantir que as atualizações feitas por um administrador de banco de dados malicioso (ou alguém que consiga obter a senha do administrador) estarão na trilha de auditoria? Explique sua resposta.

8.22 Os hackers podem ser capazes de enganá-lo a acreditar que o site é realmente um site (como de banco ou cartão de crédito) em que você confia. Isso pode ser feito extraviando e-mail, ou ainda entrando na infra-estrutura da rede e redirecionando o tráfego destinado, digamos, a `mybank.com`, para o site dos hackers. Se você informar seu nome de usuário e senha no site dos hackers, o site poderá registrar isso e usá-lo mais tarde para entrar na sua conta no site real. Quando você usa um URL como `https://mybank.com`, o protocolo HTTPS é usado para impedir tais ataques. Explique como o protocolo poderia utilizar certificados digitais para verificar a autenticidade do site.

8.23 Explique o que é um sistema de desafio-resposta para autenticação. Por que ele é mais seguro do que um sistema tradicional baseado em senha?

Sugestões de projeto

Cada um dos seguintes é um projeto grande, que pode ser um projeto de um semestre feito por um grupo de alunos. A dificuldade do projeto pode ser facilmente ajustada pela inclusão ou remoção de recursos.

Projeto 8.1 Considere o esquema E-R do Exercício prático 6.4 (Capítulo 6), que representa informações sobre os times em uma liga. Projete e implemente um sistema baseado na Web para inserir, atualizar e exibir os dados.

Projeto 8.2 Projete e implemente um sistema de carrinho de compras que permite que os compradores juntem itens em um carrinho (você pode decidir que informações devem ser fornecidas para cada item) e façam a compra conjunta. Você pode estender e usar o esquema E-R do Exercício 6.21 do Capítulo 6. É preciso verificar a disponibilidade do item e lidar com itens não disponíveis como você achar mais apropriado.

Projeto 8.3 Projete e implemente um sistema baseado na web para registrar as informações de registro e nota do aluno para os cursos em uma universidade.

Projeto 8.4 Projete e implemente um sistema que permite o registro de informações de desempenho do curso – especificamente, as notas dadas a cada aluno em cada trabalho ou exame de um curso, e o cálculo de uma soma (ponderada) das notas, para chegar às notas finais do curso. O número de trabalhos/exames não deverá ser predefinido; ou seja, mais trabalhos/exames poderão ser acrescentados a qualquer momento. O sistema também deverá admitir a conceituação, permitindo que limites sejam especificados para diversos conceitos.

Você também pode querer integrá-lo ao sistema de registro de alunos do Projeto 8.3 (talvez sendo implementado por outra equipe de projeto).

Projeto 8.5 Projete e implemente um sistema baseado na Web para reservar salas de aula na sua universidade. A reserva periódica (dias/horários fixos a cada semana para um semestre inteiro) deverá ser aceita. O cancelamento de palestras específicas em uma reserva periódica também deverá ser aceito.

Você também pode querer integrá-lo ao sistema de registro de alunos do Projeto 8.3 (talvez sendo implementado por outra equipe de projeto), para que as salas de aula possam ser reservadas para cursos, e cancelamentos de uma palestra ou palestras extras possam ser anotados em uma única interface, sendo refletidos na reserva da sala de aula e comunicados aos alunos por e-mail.

Projeto 8.6 Projete e implemente um sistema para gerenciar testes de múltipla escolha on-line. Você deverá admitir a contribuição distribuída de perguntas (por assistentes de ensino, por exemplo), a edição de perguntas por quem estiver encarregado do curso e a criação de testes do conjunto de perguntas disponível. Você também deverá ser capaz de administrar testes on-line, seja em um horário fixo para todos os alunos, ou a qualquer momento, mas com um limite de tempo do começo ao fim (um ou ambos).Dê retorno aos alunos sobre suas notas ao final do tempo reservado.

Projeto 8.7 Projete e implemente um sistema para gerenciar o serviço de cliente de e-mail. A correspondência que chega vai para um pool comum. Há um conjunto de agentes de serviço do cliente que responde ao e-mail. Se o e-mail fizer parte de uma série contínua de respostas (rastreadas por meio do campo in-reply-to), ele deverá ser respondido de preferência pelo mesmo agente que respondeu anteriormente. O sistema deverá rastrear todo o e-mail que chega e as respostas, para que um agente possa ver o histórico das perguntas de um cliente antes de responder a uma mensagem.

Projeto 8.8 Projete e implemente um mercado eletrônico simples, em que os itens podem ser listados para venda ou para compra sob várias categorias (que deverão formar uma hierarquia). Você também pode querer dar suporte a serviços de alerta, em que um usuário pode registrar o interesse em item de uma categoria em particular, talvez também com outras restrições, sem anunciar publicamente seu interesse, e é notificado quando esse item for listado para venda.

Projeto 8.9 Projete e implemente um sistema de grupo de notícias baseado na Web. Os usuários devem ser capazes de se inscrever em grupos de notícias, e navegar por artigos nos grupos. O sistema acompanha quais artigos foram lidos por um usuário, para que não sejam exibidos novamente. Ofereça também uma pesquisa de artigos antigos.

Você também pode querer oferecer um serviço de avaliação para os artigos, de modo que aqueles com maior avaliação sejam destacados, permitindo que o leitor ocupado pule os artigos com avaliação inferior.

Projeto 8.10 Projete e implemente um sistema baseado na Web para gerenciar uma "escada" de esportes. Muitas pessoas se registram e podem receber alguma avaliação inicial (talvez com base no desempenho passado). Qualquer um pode desafiar outro para uma disputa, e as avaliações são ajustadas de acordo com o resultado.

Um sistema simples para ajustar as avaliações move o vencedor para a frente do perdedor na ordem da avaliação, caso o vencedor estivesse atrás. Você pode tentar inventar sistemas de ajuste de avaliação mais complicados.

Projeto 8.11 Projete e implemente um serviço de listagem de publicação. O serviço deverá permitir a entrada de informações sobre publicações, como título, autores, ano, onde a publicação apareceu e páginas. Os autores deverão ser uma entidade separada com atributos como nome, instituição, departamento, e-mail, endereço e home page.

Sua aplicação deverá admitir várias visões sobre os mesmos dados. Por exemplo, você deverá oferecer todas as publicações por determinado autor (classificadas por ano, por exemplo), ou todas as publicações por autores de determinada instituição ou departamento. Você também deverá admitir a busca por palavras-chave, no banco de dados geral e também dentro de cada uma das views.

Projeto 8.12 Uma tarefa comum em qualquer organização é coletar informações estruturadas de um grupo de pessoas. Por exemplo, um gerente pode ter de pedir aos funcionários para entrar com seus planos de férias, um professor pode querer reunir feedback sobre um assunto em particular de alunos, ou um aluno organizando um evento pode querer permitir que outros alunos se registrem para o evento, ou alguém pode querer realizar uma votação on-line sobre algum assunto.

Crie um sistema que permitirá que os usuários criem facilmente eventos de coleta de informações. Ao criar um evento, seu criador precisa definir quem é elegível a participar; para isso, seu sistema precisa manter informações do usuário e permitir predicados definindo um subconjunto dos usuários. O criador do evento deverá ser capaz de especificar um conjunto de entradas (com tipos, valores-padrão e verificações de validação) que os usuários terão de fornecer. O evento deverá ter um prazo associado, e a capacidade de enviar lembretes aos usuários que não submeteram suas informações. O criador do evento pode receber a opção de imposição automática do prazo com base em uma data/hora especificada, ou pode decidir fazer o login e declarar que o prazo acabou. Estatísticas sobre as submissões devem ser geradas – para isso, o criador do evento pode ter permissão de criar resumos simples sobre a informação inserida. O criador do evento pode decidir tornar alguns dos resumos públicos, visualizáveis por todos os usuários, seja continuamente (por exemplo, quantas pessoas responderam) ou depois do prazo (por exemplo, qual foi a avaliação média de feedback).

Projeto 8.13 Crie uma biblioteca de funções para simplificar a criação de interfaces Web. Você precisa implementar pelo menos as seguintes funções: uma função para exibir um conjunto de resultados JDBC (com formatação tabular), funções para criar diferentes tipos de entradas de texto e numérica (com critérios de validação como tipo de entrada e intervalo opcional, impostos no cliente pelo código JavaScript apropriado), funções para entrada de valores de data e hora (com valores-padrão) e funções para criar itens de menu baseados em um conjunto de resultados. Para um crédito extra, permita que o usuário defina parâmetros de estilo como cores e fontes, e ofereça suporte de paginação nas tabelas (parâmetros de formulário ocultos podem ser usados para especificar qual página deve ser exibida). Monte uma aplicação de banco de dados de exemplo para ilustrar o uso dessas funções.

Projeto 8.14 Projete e implemente um sistema de calendário multiusuário baseado na Web. O sistema precisa acompanhar compromissos para cada pessoa, com eventos de ocorrência múltipla, como reuniões semanais, eventos compartilhados (onde uma atualização feita pelo criador do evento é refletida em todos aque-

les que compartilham o evento). Ofereça interfaces para agendar eventos multiusuários, em que um criador de evento pode acrescentar uma série de usuários que são convidados para o evento. Ofereça notificação de eventos por e-mail. Para obter créditos extras, implemente um serviço Web que possa ser usado por um programa de lembrete rodando na máquina cliente.

Notas bibliográficas

Informações sobre servlets, incluindo tutoriais, especificações de padrões e software, estão disponíveis em `java.sun.com/products/servlet`. Informações sobre JSP estão disponíveis em `java.sun.com/products/jsp`. Informações sobre bibliotecas de tags JSP também podem ser encontradas nesse URL. Informações sobre a estrutura .NET e sobre desenvolvimento de aplicações Web usando ASP.NET podem ser encontradas em `msdn.microsoft.com`.

As propostas SQL originais para declarações e triggers são discutidas em Astrahan *et al.* [1976], Chamberlin *et al.* [1976] e Chamberlin *et al.* [1981]. Melton e Simon [2001], Melton[2002], e Eisenberg e Melton [1999] oferecem um livro-texto sobre a SQL:1999, incluindo a abordagem sobre declarações e triggers em SQL:1999.

Mais informações sobre o Virtual Private Database (VPD) da Oracle, que oferece autorização detalhada entre outros recursos, poderão ser encontradas em `www.oracle.com/technology/deploy/security/index.html`. A autorização detalhada também é discutida em Rizvi *et al.* [2004].

Atreva e outros [2002] oferecem um livro-texto sobre as assinaturas digitais, incluindo certificados digitais X.509 e infra-estrutura de chave pública. Informações sobre o sistema de logon único Pubcookie podem ser encontradas em `www.pubcookie.org`.

Ferramentas

O desenvolvimento de uma aplicação Web requer várias ferramentas de software, como um servidor de aplicações, um compilador e um editor para uma linguagem de programação como Java ou C#, e outras ferramentas opcionais como um servidor Web.

Listamos algumas das ferramentas mais conhecidas aqui: o Java SDK da Sun (`java.sun.com`), o sistema Apache Tomcat (`jakarta.apache.org`), que admite servlets e JSP, o servidor Web Apache (`apache.org`), o servidor de aplicações JBoss (`jboss.org`), as ferramentas ASP.NET da Microsoft (`msdn.microsoft.com/asp.net/`), IBM WebSphere (`www.software.ibm.com`), Resin da Caucho (`www.caucho.com`), produtos Coldfusion e JRun da Allaire (`www.allaire.com`) e Zope (`www.zope.org`). Algumas dessas, como o Apache Tomcat e o servidor Web Apache, são gratuitas para qualquer uso, algumas são gratuitas para uso não comercial ou para uso pessoal, enquanto outras precisam ser pagas. Veja mais informações nos respectivos sites.

PARTE 3

Bancos de dados baseados em objeto e XML

Várias áreas de aplicação para os sistemas de banco de dados são limitadas pelas restrições do modelo de dados relacional. Conseqüentemente, os pesquisadores têm desenvolvido vários modelos de dados baseados em um método orientado a objeto, para lidar com esses domínios de aplicação.

O modelo relacional de objeto, descrito no Capítulo 9, combina recursos dos modelos relacional e orientado a objeto. Esse modelo fornece o sistema de tipo rico das linguagens orientadas a objeto, combinado com as relações, como a base para o armazenamento de dados. Ele aplica herança às relações, não apenas aos tipos. O modelo de dados relacional de objeto fornece um caminho de migração estável dos bancos de dados relacionais, que é atraente para os fornecedores de banco de dados relacional. Como resultado, o padrão SQL:1999 inclui diversos recursos orientados a objeto no seu sistema de tipos, enquanto continua a usar o modelo relacional como o modelo básico.

O termo banco de dados orientado a objeto é usado para descrever um sistema de banco de dados que aceita acesso direto aos dados das linguagens de programação orientadas a objeto, sem exigir uma linguagem de consulta relacional como interface de banco de dados. O Capítulo 9 também fornece um breve resumo dos bancos de dados orientados a objeto.

A linguagem XML foi inicialmente projetada como uma forma de acrescentar informações de marcação a documentos de texto, mas se tornou importante devido a suas aplicações na troca de dados. A XML oferece uma maneira de representar dados que possuem estrutura aninhada, além de permitir uma grande flexibilidade na estruturação dos dados, o que é importante para certos tipos de dados não tradicionais. O Capítulo 10 descreve a linguagem XML e, depois, apresenta diferentes maneiras de expressar consultas em dados representados na XML, incluindo a linguagem de consulta XML XQuery, que está ganhando ampla aceitação e uso.

CAPÍTULO 9

Bancos de dados baseados em objeto

As aplicações tradicionais de banco de dados consistem em tarefas de processamento de dados, como transações bancárias e gerenciamento de folha de pagamento, com tipos de dados relativamente simples, que são adequados ao modelo de dados relacional. Uma vez que os sistemas de banco de dados foram aplicados a uma faixa mais ampla de aplicações, como projeto auxiliado por computador e sistemas de informações geográficas, as limitações impostas pelo modelo relacional se apresentaram como um obstáculo. A solução era a introdução dos bancos de dados baseados em objeto, que permitem lidar com tipos de dados complexos.

Visão geral

O primeiro obstáculo enfrentado pelos programadores usando o modelo de dados relacional era o sistema de tipo limitado aceito pelo modelo relacional. Os domínios de aplicação complexos exigem tipos de dados correspondentemente complexos, como estruturas de registros aninhados, atributos e herança multivalores, que são aceitos pelas linguagens de programação tradicionais. Esses recursos são, na verdade, aceitos nas notações E-R e E-R estendida, mas tiveram de ser traduzidos para tipos de dados SQL mais simples. O **modelo de dados relacional de objeto** estende o modelo de dados relacional fornecendo um sistema de tipo mais rico, incluindo tipos de dados complexos e orientação a objeto. As linguagens de consulta relacionais, em especial a SQL, também precisam ser estendidas para lidar com o sistema de tipo mais rico. Essas extensões tentam preservar as fundações relacionais – sobretudo o acesso declarativo aos dados – enquanto estende a capacidade de modelagem. Os **sistemas de banco de dados relacionais**, ou seja, os sistemas de banco de dados baseados no modelo de relação de objeto, fornecem um caminho de migração conveniente para os usuários dos bancos de dados relacionais que desejam usar recursos orientados a objeto.

O segundo obstáculo era a dificuldade de acessar dados de banco de dados a partir de programas escritos em linguagens de programação como C++ ou Java. Apenas estender o sistema de tipo aceito pelo banco de dados não era suficiente para resolver esse problema completamente.

As diferenças entre o sistema de tipo do banco de dados e o sistema de tipo da linguagem de programação tornam o armazenamento e a recuperação de dados mais complicados e precisam ser minimizadas. A necessidade de expressar acesso a banco de dados usando uma linguagem (SQL) que é diferente da linguagem de programação novamente dificulta o trabalho do programador. Para muitas aplicações, é desejável ter construções ou extensões de linguagem de programação que permitam acesso direto a dados no banco de dados, sem ter de usar uma linguagem intermediária como SQL.

O termo **linguagens de programação persistentes** se refere às extensões de linguagens de programação existentes para acrescentar persistência e outros recursos de banco de dados, usando o sistema de tipo nativo da linguagem de programação. O termo **sistemas de banco de dados orientados a objeto** é usado para se referir aos sistemas de banco de dados que aceitam um sistema de tipo orientado a objeto e permitem acesso direto aos dados de uma linguagem de programação orientada a objeto usando o sistema de tipo nativo da linguagem.

Neste capítulo, vamos primeiramente explicar a motivação para o desenvolvimento de tipos de dados complexos. Depois, estudaremos os sistemas de banco de dados relacionais de objeto. Nossa análise é baseada nas extensões re-

lacionais de objeto acrescentadas na versão SQL:1999 do padrão SQL. Nossa descrição é baseada no padrão SQL, especificamente usando recursos que foram introduzidos na SQL:1999 e SQL:2003. Note que a maioria dos produtos de banco de dados aceita apenas um subconjunto dos recursos SQL descritos aqui. Consulte o manual do usuário do sistema de banco de dados que você usa para descobrir quais recursos ele aceita.

Em seguida, estudaremos brevemente os sistemas de banco de dados orientados a objeto que acrescentam suporte de persistência às linguagens de programação orientadas a objeto. Finalmente, mostraremos situações em que o método relacional de objeto é melhor do que o método orientado a objeto e vice-versa, e mencionaremos critérios para escolher entre eles.

Tipos de dados complexos

As aplicações de banco de dados tradicionais consistem em tarefas de processamento de dados, como transações bancárias e gerenciamento de folha de pagamento. Essas aplicações possuem conceitualmente tipos de dados simples. Os itens de dados básicos são registros muito pequenos e cujos campos são atômicos – isto é, não são mais estruturados e a primeira forma normal se aplica (veja o Capítulo 7). Além disso, existem apenas alguns tipos de registro.

Nos últimos anos, aumentou a necessidade de métodos para lidar com tipos de dados mais complexos. Considere, por exemplo, os endereços. Embora um endereço completo pudesse ser visto como um item de dados atômico de tipo string, essa visão ocultava detalhes como a rua, cidade, estado e código postal, que poderia ser de interesse para consultas. Por outro lado, se um endereço fosse representado dividindo-o em seus componentes (rua e número, cidade, estado e código postal), escrever consultas seria mais complicado, já que elas precisariam mencionar cada campo. Uma alternativa melhor é permitir tipos de dados estruturados, que permitem um tipo *endereço* com subpartes *rua_número*, *cidade*, *estado* e *cep*.

Como outro exemplo, considere os atributos multivalores do modelo E-R. Esses atributos são naturais, por exemplo, para representar números de telefone, já que as pessoas podem ter mais de um telefone. A alternativa de normalização criando uma nova relação é cara e artificial para esse exemplo.

Com sistemas de tipo complexos, podemos representar diretamente conceitos do modelo E-R, como atributos compostos, atributos multivalores, generalização e especialização, sem uma tradução complexa para o modelo relacional.

No Capítulo 7, definimos a *primeira forma normal* (1FN), que exige que todos os atributos tenham *domínios atômicos*. Lembre-se de que um domínio é *atômico* se os elementos do domínio são considerados unidades indivisíveis.

A suposição da 1FN é natural nos exemplos de banco que consideramos. Entretanto, nem todas as aplicações são mais bem modeladas pelas relações na 1FN. Por exemplo, em vez de ver um banco de dados como um conjunto de registros, os usuários de certas aplicações o vêem como um conjunto de objetos (ou entidades). Esses objetos podem exigir vários registros para sua representação. Uma interface simples e fácil de usar requer uma correspondência um-para-um entre a noção intuitiva do usuário de um objeto e a noção do sistema de banco de dados de um item de dados.

Considere, por exemplo, uma aplicação de biblioteca e suponha que desejamos armazenar as seguintes informações para cada livro:

- Título
- Lista de autores
- Editora
- Conjunto de palavras-chaves

Podemos ver que, se definirmos uma relação para essas informações, vários domínios serão não atômicos.

- **Autores**. Um livro pode ter uma lista de autores, que podemos representar como um array. Entretanto, podemos querer encontrar todos os livros dos quais Jones foi um dos autores. Portanto, estamos interessados em uma subparte do elemento domínio "autores".
- **Palavras-chave**. Se armazenarmos um conjunto de palavras-chave para um livro, esperamos ser capazes de recuperar todos os livros cujas palavras-chave incluem uma ou mais delas. Portanto, vemos o domínio do conjunto de palavras-chave como não atômico.

Título	*array_autor*	*editora*	*conjunto_palavras_chave*
		(*nome*, *divisão*)	
Compiladores	{Smith, Jones}	(McGraw-Hill, New York)	{analise, parsing}
Redes	{Jones, Frick}	(Oxford, London)	{Internet, Web}

Figura 9.1 Relação de livros não 1FN, *livros*.

Título	*autor*	*posição*
Compiladores	Smith	1
Compiladores	Jones	2
Redes	Jones	1
Redes	Frick	2

autores

Título	*palavra_chave*
Compiladores	parsing
Compiladores	análise
Redes	Internet
Redes	Web

palavras-chaves

Título	*nome_editora*	*divisão_editora*
Compiladores	McGraw-Hill	New York
Redes	Oxford	London

livros4

Figura 9.2 Versão 4FN da relação *livros*.

- **Editora**: Diferente de *palavras-chaves* e de *autores*, *editora* não tem um domínio com valor de conjunto. No entanto, podemos ver *editora* como consistindo nos subcampos *nome* e *divisão*. Essa visão torna o domínio de *editora* não atômico.

A Figura 9.1 mostra uma relação de exemplo, *livros*.

Por simplicidade, consideramos que o título de um livro identifica o livro unicamente.[1] Podemos, então, representar as mesmas informações usando o seguinte esquema:

- *autores*(*título, autor, posição*)
- *palavras-chave*(*título, palavra-chave*)
- *livros4*(*título, nome_editora, divisão_editora*)

Esse esquema satisfaz a quarta forma normal (4FN). A Figura 9.2 mostra a representação normalizada dos dados da Figura 9.1.

Embora nosso banco de dados de livro de exemplo possa ser adequadamente expresso sem usar relações aninhadas, o uso dessas relações leva a um modelo mais fácil de entender. O usuário ou programador típico de um sistema de recuperação de informações pensa no banco de dados em termos de livros tendo conjuntos de autores, como os modelos de projeto não 1FN. O projeto 4FN exige que as consultas usem junções de várias relações, enquanto o projeto não 1FN torna muitos tipos de consultas mais fáceis.

Por outro lado, pode ser melhor usar uma representação de primeira forma normal em vez de coleções em outras situações. Por exemplo, considere a relação *depositante* em nosso exemplo de banco. A relação é muitos-para-muitos entre *clientes* e *contas*. Poderíamos armazenar um conjunto de contas com cada cliente, ou um conjunto de clientes com cada conta, ou ambos. Se armazenarmos ambos, poderíamos ter redundância de dados (a relação de um determinado cliente com uma conta específica seria armazenada duas vezes).

A capacidade de usar tipos de dados complexos como conjuntos e arrays pode ser útil em muitas aplicações mas deve ser usada com cuidado.

Tipos estruturados e herança em SQL

Antes do SQL:1999, o sistema de tipo SQL consistia em um conjunto bastante simples de tipos predefinidos. O SQL:1999 acrescentou um sistema de tipo extensivo a SQL, permitindo tipos estruturados e herança de tipo.

Tipos estruturados

Os tipos estruturados permitem que atributos de diagramas E-R sejam representados diretamente. Por exemplo, pode-

1. Esta suposição não se mantém na realidade. Os livros normalmente são identificados por um número de ISBN de 10 dígitos que identifica unicamente cada livro publicado.

mos definir o seguinte tipo estruturado para representar um atributo composto *nome* com o atributo componente *prenome* e *sobrenome*:

```
create type Nome as
  (prenome varchar(20),
  sobrenome varchar(20))
  final
```

Da mesma forma, o seguinte tipo estruturado pode ser usado para representar um atributo composto *endereço*:

```
create type Endereço as
  (rua varchar(20),
  cidade varchar(20)
  cep varchar(9))
  not final
```

Esses tipos são chamados de tipos **definidos pelo usuário** na SQL. Essa definição corresponde ao diagrama E-R na Figura 6.4. As especificações **final** e **not final** estão relacionadas à subtipagem, que descreveremos mais adiante, na próxima seção.[2]

Agora, podemos usar esses tipos para criar atributos compostos em uma relação, simplesmente declarando um atributo para ser de um desses tipos. Por exemplo, poderíamos criar uma tabela *cliente* desta maneira:

```
create table cliente(
  nome Nome,
  endereço Endereço,
  dataDeNascimento date)
```

Os componentes de um atributo composto podem ser acessados usando uma notação de "ponto"; por exemplo, *nome.prenome* retorna o componente *prenome* do atributo *nome*. Um acesso ao atributo *nome* retornaria um valor do tipo estruturado *Nome*.

Também podemos criar uma tabela cujas linhas são de um tipo definido pelo usuário. Por exemplo, podemos definir um tipo *TipoCliente* e criar a tabela *cliente* desta maneira:

```
create type TipoCliente as(
  nome Nome,
  endereço Endereço,
  dataDeNascimento date)
  not final
create table cliente of TipoCliente
```

Um modo alternativo de definir atributos compostos em SQL é usar **tipos de linha** não nomeados. Por exemplo, a relação representando informações de cliente poderiam ter sido criados usando tipos de linha desta forma:

```
create table cliente_r(
  nome row (prenome varchar(20),
      sobrenome varchar(20))
  endereço row (rua varchar(20),
      cidade varchar(20)
      cep varchar(9)),
  dataDeNascimento date)
```

Essa definição é equivalente à definição de tabela anterior, exceto que os atributos *nome* e *endereço* possuem tipos não nomeados, e as linhas da tabela também possuem um tipo não nomeado.

A consulta a seguir ilustra como acessar atributos de componente de um atributo composto. A consulta encontra o sobrenome e a cidade de cada cliente.

```
select nome.sobrenome, endereço.cidade
from cliente
```

Um tipo estruturado pode ter **métodos** definidos nele. Declaramos métodos como parte da definição de tipo de um tipo estruturado:

```
create type TipoCliente as(
  nome Nome,
  endereço Endereço,
  dataDeNascimento date)
  not final
method idadeNaData(naData date)
  return interval year
```

Podemos criar o corpo do método separadamente:

```
create instance method idadeNaData(naData
date)
  return interval year
  for TipoCliente
begin
  return naData – self.dataDeNascimento;
end
```

Observe que a cláusula **for** indica para que tipo é esse método, enquanto a palavra-chave **instance** indica que esse método é executado em uma instância do tipo *Cliente*. A variável **self** se refere à instância *Cliente* em que o método é chamado. O corpo do método pode conter instruções procedurais, que vimos anteriormente na seção "Funções e construções procedurais" do Capítulo 4. Os

2. A especificação final para *Nome* indica que não podemos criar subtipos para *nome*, enquanto a especificação not final para *Endereço* indica que podemos criar subtipos para *endereço*.

métodos podem atualizar os atributos da instância em que são executados.

Os métodos podem ser chamados em instâncias de um tipo. Se tivéssemos criado uma tabela *cliente* do tipo *TipoCliente*, poderíamos chamar o método *idadeNaData( )* como ilustrado a seguir, para encontrar a idade de cada cliente.

```
select nome.sobrenome,
idadeNaData(current_date)
from cliente
```

Na SQL:1999, as **funções construtoras** são usadas para criar valores de tipos estruturados. Uma função com o mesmo nome de um tipo estruturado é uma função construtora para o tipo estruturado. Por exemplo, poderíamos declarar uma função construtora para o tipo *Nome* da seguinte forma:

```
create function Nome (prenome varchar(20),
sobrenome varchar(20))
return Nome
begin
  set self.prenome = prenome;
  set self.sobrenome = sobrenome;
end
```

Podemos, então, usar **new** *Nome*('John', 'Smith') para criar um valor do tipo *Nome*.

Podemos construir um valor de linha listando seus atributos entre parênteses. Por exemplo, se declararmos um atributo *nome* como um tipo de linha com componentes *prenome* e *sobrenome*, podemos construir este valor para ele:

```
('Ted', 'Codd')
```

sem usar uma função construtora.

Como padrão, todo tipo estruturado tem uma construtora sem argumentos, que define os atributos em seus valores-padrão. Quaisquer outras construtoras precisam ser criadas explicitamente. Pode haver mais de uma construtora para o mesmo tipo estruturado; embora tenham o mesmo nome, elas precisam ser distinguíveis pelo número de argumentos e tipos de seus argumentos.

A seguinte instrução ilustra como podemos criar uma nova tupla na relação *Cliente*. Consideramos que uma construtora foi definida para *Endereço*, exatamente como a construtora que definimos para *Nome*.

```
insert into Cliente
  values
    (new Nome('John', 'Smith'),
     new Endereço('20 Main St', 'New York',
'11001'),
     date '1960-8-22')
```

Herança de tipo

Suponha que temos a seguinte definição de tipo para pessoas:

```
create type Pessoa
  (nome varchar(20),
  endereço varchar(20))
```

Podemos querer armazenar informações extras no banco de dados sobre pessoas que são estudantes e sobre pessoas que são professores. Como estudantes e professores são pessoas, podemos usar herança para definir os tipos estudante e professor na SQL:

```
create type Estudante
  under Pessoa
  (grau varchar(20),
  departamento varchar(20))
create type Professor
  under Pessoa
  (salário integer(20),
  departamento varchar(20))
```

Tanto *Estudante* quanto *Professor* herdam os atributos de *Pessoa* – ou seja, *nome* e *endereço*. Dizemos que *Estudante* e *Professor* são subtipos de *Pessoa*, e *Pessoa* é um supertipo de *Estudante*, assim como de *Professor*.

Os métodos de um tipo estruturado são herdados por seus subtipos, assim como os atributos. Entretanto, um subtipo pode redefinir o efeito de um método declarando o método novamente, usando **overriding method** no lugar de **method** na declaração do método.

O padrão SQL também exige um campo extra no final da definição de tipo, cujo valor é **final** ou **not final**. A palavra-chave **final** diz que subtipos podem não ser criados do tipo dado, enquanto **not final** diz que subtipos podem ser criados.

Agora, suponha que queremos armazenar informações sobre professores assistentes, que são simultaneamente professores e estudantes, talvez até em departamentos diferentes. Podemos fazer isso se o sistema de tipo aceitar **herança múltipla**, em que um tipo é declarado como um subtipo de múltiplos tipos. Note que o padrão SQL (até as versões SQL:1999 e SQL:2003 pelo menos) não aceita herança múltipla, embora futuras versões do padrão SQL podem aceitá-la.

Por exemplo, se nosso sistema de tipo aceitar herança múltipla, poderemos definir um tipo para professor assistente desta maneira:

```
create type ProfessorAssistente
  under Estudante, Professor
```

ProfessorAssistente herdaria todos os atributos de *Estudante* e *Professor*. Entretanto, existe um problema, já que os atributos *nome*, *endereço* e *departamento* estão presentes em *Estudante* e também em *Professor*.

Os atributos *nome* e *endereço* são, na realidade, herdados de uma origem comum, *Pessoa*. Portanto, não há conflito causado por herdá-los de *Estudante*, bem como de *Professor*. Contudo, o atributo *departamento* é definido separadamente em *Estudante* e *Professor*. Na verdade, um professor assistente pode ser um estudante de um departamento e um professor de outro departamento. Para evitar um conflito entre as duas ocorrências de *departamento*, podemos renomeá-lo usando uma cláusula **as**, como nesta definição do tipo *ProfessorAssistente*:

```
create type ProfessorAssistente
    under Estudante with (departamento as
depto_estudante),
        Professor with (departamento as depto_professor)
```

Observe novamente que a SQL aceita apenas herança única – ou seja, um tipo pode herdar apenas de um único tipo; a sintaxe usada é como em nossos exemplos anteriores. A herança múltipla com no exemplo do *ProfessorAssistente* não é aceita na SQL.

Na SQL, como na maioria das outras linguagens, um valor de um tipo estruturado precisa ter exatamente um "tipo mais específico". Isto é, cada valor precisa ser associado a um tipo específico – chamado o seu **tipo mais específico** – quando ele é criado. Por meio de herança, ele também é associado a cada um dos supertipos do seu tipo mais específico. Por exemplo, suponha que uma entidade possui o tipo *Pessoa*, bem como o tipo *Estudante*. Então, o tipo mais específico da entidade é *Estudante*, já que *Estudante* é um subtipo de *Pessoa*. Entretanto, uma entidade não pode ter o tipo *Estudante* e o tipo *Professor* a menos que tenha um tipo, como *ProfessorAssistente*, que seja um subtipo de *Professor*, bem como de *Estudante* (o que não é possível na SQL, já que a herança múltipla não é aceita pela SQL).

Herança de tabela

As subtabelas na SQL correspondem à noção de E-R de especialização/generalização. Por exemplo, suponha que definimos a tabela *pessoas* da seguinte maneira:

```
create table pessoas of Pessoa
```

Podemos, então, definir as tabelas *estudantes* e *professores* como **subtabelas** de *pessoas*, desta forma:

```
create table estudantes of Estudante
    under pessoas
create table professores of professor
    under pessoas
```

Os tipos das subtabelas precisam ser subtipos do tipo da tabela-pai. Portanto, todo atributo presente em *pessoas* também está presente nas subtabelas.

Além disso, quando declaramos *estudantes* e *professores* como subtabelas de *pessoas*, cada tupla presente em *estudantes* ou *professores* também se torna implicitamente presente em *pessoas*. Assim, se uma consulta usar a tabela *pessoas*, ela não encontrará apenas tuplas diretamente inseridas nessa tabela, mas também tuplas inseridas em suas subtabelas (*estudantes* e *professores*). Entretanto, apenas os atributos que estão presentes em *pessoas* podem ser acessados por essa consulta.

A SQL permite encontrar tuplas que estejam em *pessoas*, mas não em suas subtabelas, usando "**only** *pessoas*" no lugar de *pessoas* em uma consulta. A palavra-chave **only** também pode ser usada nas instruções delete e update. Sem a palavra-chave **only**, uma instrução delete em uma supertabela, como *pessoas*, também exclui tuplas que foram originalmente inseridas nas subtabelas (como *estudantes*); por exemplo, uma instrução

```
delete from pessoas where P
```

excluiria todas as tuplas da tabela *pessoas* (bem como suas subtabelas *estudantes* e *professores*) que satisfazem *P*. Se a palavra-chave **only** for incluída na nessa instrução, as tuplas que foram inseridas nas subtabelas não são afetadas, ainda que satisfaçam as condições da cláusula **where**. Consultas subseqüentes na supertabela continuariam a encontrar essas tuplas.

Conceitualmente, assim como com os tipos, a herança múltipla também é possível com as tabelas. Por exemplo, podemos criar uma tabela do tipo *ProfessorAssistente*:

```
create table professores_assistentes
of ProfessorAssistente
    under estudantes, professores
```

Como resultado da declaração, cada tupla presente na tabela *professores_assistentes* também está implicitamente presente na tabela *professores* e na tabela *estudantes* e, por sua vez, na tabela *pessoas*. Devemos observar, no entanto, que a herança múltipla de tabelas não é aceita pela SQL.

Existem alguns requisitos de consistência para subtabelas. Antes de expormos as restrições, precisamos de uma definição: dizemos que as tuplas em uma subtabela correspondem às tuplas em uma tabela-pai se elas tiverem

os mesmos valores para todos os atributos herdados. Portanto, as tuplas correspondentes representam a mesma entidade.

Os requisitos de consistência para subtabelas são:

1. Cada tupla da supertabela pode corresponder a, no máximo, uma tupla em cada uma de suas subtabelas imediatas.
2. A SQL tem uma restrição adicional de que todas as tuplas correspondentes a cada uma das outras precisam ser derivadas de uma tupla (inserida na tabela).

Por exemplo, sem a primeira condição, poderíamos ter duas tuplas em *estudantes* (ou *professores*) que correspondem à mesma pessoa.

A segunda condição descarta uma tupla em *pessoas* que seja correspondente tanto a uma tupla em *estudantes* quanto a uma tupla em *professores*, a menos que todas essas tuplas estejam implicitamente presentes porque uma tupla foi inserida em uma tabela *professores_assistentes*, que é uma subtabela de *professores* e *estudantes*.

Como a SQL não aceita herança múltipla, a segunda condição, na verdade, impede que uma pessoa seja um professor e um estudante. Mesmo que a herança múltipla fosse aceita, problema idêntico surgiria se a subtabela *professores_assistentes* estivesse ausente. É claro, seria útil modelar uma situação em que uma pessoa pode ser um professor e um estudante, mesmo que uma subtabela *professores_assistentes* comum não esteja presente. Portanto, pode ser útil remover a segunda restrição de consistência. Fazer isso permitiria que um objeto tivesse vários tipos, sem exigir que ele tenha um tipo mais específico.

Por exemplo, suponha que novamente tenhamos o tipo *Pessoa*, com subtipos *Estudante* e Professor, e a tabela correspondente *pessoas*, com subtabelas *professores* e *estudantes*. Podemos, então, ter uma tupla em *professores* e uma tupla em *estudantes* correspondentes à mesma tupla em *pessoas*. Não há necessidade de ter um tipo *ProfessorAssistente* a menos que queiramos armazenar atributos extras ou redefinir métodos de uma maneira específica às pessoas que são tanto estudantes quanto professores.

Note, entretanto, que a SQL infelizmente proíbe essa situação, devido ao requisito de consistência 2. Como a SQL também não aceita herança múltipla, não podemos usar herança para modelar uma situação em que uma pessoa pode ser tanto um estudante quanto um professor. Como resultado, as subtabelas SQL não podem ser usadas para representar especializações sobrepostas do modelo E-R.

É claro, podemos criar tabelas separadas para representar as especializações/generalizações sobrepostas sem usar herança. O processo foi descrito anteriormente, na seção "Representação da generalização" do Capítulo 6. Nesse exemplo, criaríamos tabelas *pessoas*, *estudantes* e *professores*, com as tabelas *estudantes* e *professores* contendo o atributo de chave primária de *Pessoa* e outros atributos específicos a *Estudante* e *Professor*, respectivamente. A tabela *pessoas* conteria informações sobre todas as pessoas, incluindo estudantes e professores. Então, precisaríamos incluir restrições de integridade referencial apropriadas para garantir que os estudantes e professores também sejam representados na tabela *pessoas*.

Em outras palavras, podemos criar nossa própria implementação do mecanismo de subtabela usando recursos existentes da SQL, com algum trabalho extra em definir a tabela, bem como algum trabalho extra em tempo de consulta para especificar junções para acessar atributos necessários.

Para terminar a seção, devemos notar que a SQL define um novo privilégio chamado **under**, que é necessário para criar um subtipo ou uma subtabela sob outro tipo ou tabela. A motivação para esse privilégio é semelhante à motivação para o privilégio **references**.

Tipos array e multiconjunto na SQL

A SQL aceita dois tipos de coleção: arrays e multiconjuntos. Os tipos array foram incluídos na SQL:1999, enquanto os tipos multiconjunto foram incluídos na SQL:2003. Lembre-se de que um *multiconjunto* é uma coleção desordenada, em que um elemento pode ocorrer mais de uma vez. Os multiconjuntos são como os conjuntos, exceto que um conjunto permite que cada elemento ocorra no máximo uma vez.

Suponha que desejamos registrar informações sobre livros, incluindo um conjunto de palavras-chave para cada livro. Suponha também que desejássemos armazenar os nomes dos autores de um livro como um array; diferente dos elementos em um multiconjunto, os elementos de um array são ordenados e, portanto, podem distinguir o primeiro autor do segundo e assim por diante. O seguinte exemplo ilustra como esses atributos de array e multiconjunto podem ser definidos na SQL.

```
create type Editora as
    (nome varchar(20),
    divisão varchar(20))
create type Livro as
    (título varchar(20),
    array_autor varchar(20) array [10],
    data_pub date,
    editora Editora,
    conjunto_palavras_chave varchar(20) multiset)
create table livros of Livro
```

A primeira instrução define um tipo chamado *Editora*, que tem dois componentes: um nome e uma divisão. A segunda instrução define um tipo estruturado, *Livro*, que contém um *título* um *array_autor*, que é um array de até 10 nomes de autores, uma data de publicação, uma editora (do tipo *Editora*) e um multiconjunto de palavras-chave. Finalmente, uma tabela *livros* contendo tuplas do tipo *Livro* é criada.

Observe que usamos um array, em vez de um multiconjunto, para armazenar os nomes dos autores, já que a ordenação de autores geralmente tem alguma significância, embora acreditemos que a ordenação das palavras-chave associadas a um livro não seja significante.

Em geral, os atributos multivalores de um esquema E-R podem ser mapeados para atributos com valor de multiconjunto na SQL; se a ordenação for importante, os arrays SQL podem ser usados no lugar de multiconjuntos.

Criando e acessando valores de coleção

Um array de valores pode ser criado na SQL:1999 da seguinte maneira:

```
array['Silberschatz', 'Korth', 'Sudarshan']
```

Da mesma forma, um multiconjunto de palavras-chave pode ser construído assim:

```
multiset['computador', 'banco de dados', 'SQL']
```

Portanto, podemos criar uma tupla do tipo definido pela relação *livros* como:

```
('Compiladores', array['Smith', 'Jones'], new
Editora('McGraw-Hill', 'New York'), multiset['parsing',
'análise'])
```

Aqui, criamos um valor para o atributo *Editora* chamando uma função *construtora* para *Editora* com argumentos apropriados. Observe que essa construtora para *Editora* precisa ser criada explicitamente e não está presente como padrão; ela pode ser declarada da mesma forma que a construtora para *Nome*, que vimos anteriormente na seção "Tipos estruturados e herança em SQL".

Se quisermos inserir a tupla anterior na relação *livros*, podemos executar a instrução

```
insert into livros
values
('Compiladores', array['Smith', 'Jones'],
        new Editora('McGraw-Hill', 'New York'),
        multiset['parsing', 'análise'])
```

Podemos acessar ou atualizar elementos de um array especificando o índice do array, por exemplo, *array_autor*[1].

Consultando atributos com valor de coleção

Agora, vamos considerar como manipular atributos avaliados para coleção em consultas. Uma expressão avaliando para uma coleção pode aparecer em qualquer lugar que um nome de relação pode aparecer, como em uma cláusula **from**, como ilustram os parágrafos a seguir. Usamos a tabela *livros* que definimos anteriormente.

Se quisermos encontrar todos os livros que possuem a palavra "banco de dados" como uma de suas palavras-chave, podemos usar esta consulta:

```
select título
from livros
where 'banco de dados' in (unnest(conjunto_pala-
vras-chave))
```

Repare que usamos **unnest**(*conjunto_palavras-chave*) em uma posição onde a SQL sem relações aninhadas teria exigido uma subexpressão **select-from-where**.

Se considerarmos que um determinado livro possui três autores, podemos escrever:

```
select array_autor[1], array_autor[2], array_autor[3]
from livros
where título = 'Database System Concepts'
```

Agora, suponha que queremos uma relação contendo pares da forma "título, nome_autor" para cada livro e cada autor do livro. Podemos usar esta consulta:

```
select B.título, A.autor
from livros as B, unnest(B.array_autor) as A(autor)
```

Como o atributo *array_autor* de *livros* é um campo avaliado para coleção, **unnest**(*B.array_autor*) pode ser usado em uma cláusula **from**, em que uma relação é esperada. Note que a variável de tupla *B* é visível a essa expressão, já que ela é definida *anteriormente* na cláusula **from**.

Ao desaninhar um array, a consulta anterior perde informações sobre a ordenação dos elementos no array. A cláusula **unnest with ordinality** pode ser usada para obter essas informações, como ilustrado pela consulta a seguir. Essa consulta pode ser usada para gerar a relação *autores*, que vimos anteriormente, a partir da relação *livros*.

```
select título, A.autor, A.posição
from livros as B, unnest(B.array_autor) with ordinality
as A(autor, posição)
```

Título	*autor*	*nome_editora*	*divisão_editora*	*palavra-chave*
Compiladores	Smith	McGraw-Hill	New York	parsing
Compiladores	Jones	McGraw-Hill	New York	parsing
Compiladores	Smith	McGraw-Hill	New York	analise
Compiladores	Jones	McGraw-Hill	New York	analise
Redes	Jones	Oxford	London	Internet
Redes	Frick	Oxford	London	Internet
Redes	Jones	Oxford	London	Web
Redes	Frick	Oxford	London	Web

Figura 9.3 *livros_simples*: resultado dos atributos desaninhados *array_autor* e *conjunto_palavras_chave* da relação *livros*.

A cláusula **with ordinality** gera um atributo extra que registra a posição do elemento no array. Uma consulta semelhante, mas sem a cláusula **with ordinality**, pode ser usada para gerar a relação *palavra-chave*.

Aninhando e desaninhando

A transformação de uma relação aninhada em uma forma com menos atributos avaliados para relação (ou com nenhum deles) é chamada de **desaninhamento**. A relação *livros* possui dois atributos, *array_autor* e *conjunto_palavras_chave*, que são coleções, e dois atributos, *título* e *editora*, que não são. Suponha que queiramos converter a relação em uma única relação simples, sem relações aninhadas ou tipos estruturados como atributos. Podemos usar a seguinte consulta para realizar a tarefa:

```
select título, A.autor, editora.nome as nome_editora,
editora.divisão
  as divisão_editora, K.palavra_chave
from livros as B, unnest(B.array_autor) as A(autor,
unnest (B.conjunto_palavras_chave) as K(palavra_chave)
```

A variável *B* na cláusula **from** é declarada para variar sobre *livros*. A variável *A* é declarada para variar sobre os autores em *array_autor* para o livro *B*, e *K* é declarada para variar sobre as palavras-chave no *conjunto_palavras_chave* do livro *B*. A Figura 9.1 mostra uma relação *livros* de exemplo e a Figura 9.3 mostra a relação, que chamaremos de *livros_simples*, que é o resultado da consulta anterior. Note que a relação *livros_simples* está na primeira forma normal (1FN), já que todos os seus atributos são avaliados como atômicos.

O processo inverso de transformar uma relação na 1FN em uma relação aninhada é chamado de **aninhamento**. O aninhamento pode ser realizado por uma extensão do agrupamento na SQL. No uso normal do agrupamento na SQL, uma relação multiconjunto temporária é criada (logicamente) para cada grupo, e uma função agregada é aplicada na relação temporária para obter um único valor (atômico). A função **collect** retorna o multiconjunto de valores; em vez de criar um único valor, podemos criar uma relação aninhada. Suponha que recebemos a relação na 1FN *livros_simples*, como na Figura 9.3. A consulta a seguir aninha a relação no atributo *palavra_chave*:

```
select título, autor, Editora(nome_editora, divisão_editora)
as editora,
   collect(palavra_chave) as conjunto_palavras_chave
from livros_simples
group by título, autor, editora
```

O resultado da consulta na relação *livros_simples* da Figura 9.3 aparece na Figura 9.4.

título	*autor*	*editora*	*conjunto_palavras_chave*
		(nome_editora, divisão_editora)	
Compiladores	Smith	(McGraw-Hill, New York)	{parsing, analise}
Compiladores	Jones	(McGraw-Hill, New York)	{parsing, analise}
Redes	Jones	(Oxford, London)	{Internet, Web}
Redes	Frick	(Oxford, London)	{Internet, Web}

Figura 9.4 Uma versão parcialmente aninhada da relação *livros_simples*.

Se quisermos aninhar o atributo autor também em um multiconjunto, podemos usar a consulta:

```
select título, collect(autor) as conjunto_autores,
   Editora(nome_editora, divisão_editora) as editora,
     collect(palavra_chave) as conjunto_palavras-chave
from livros_simples
group by título, editora
```

Outro método para criar relações aninhadas é usar subconsultas na cláusula **select**. Uma vantagem do método de subconsulta é que uma cláusula **order by** pode ser usada opcionalmente na subconsulta para gerar resultados em uma ordem desejada, que, então, pode ser usada para criar um array. A consulta a seguir ilustra esse método; as palavras-chave **array** e **multiset** especificam que um array e um subconjunto, respectivamente, devem ser criados a partir dos resultados das subconjuntas.

```
select título,
   array(select autor
     from autores as A
     where A.título = B.título
order by A.position) as array_autor,
   Editora(nome_editora, divisão_editora) as editora,
   multiset(select palavra_chave
     from palavras_chave as K
     where K.título = B.título) as conjunto_palavras_chave,
from livros4 as B
```

O sistema executa as subconsultas aninhadas na cláusula **select** para cada tupla gerada pelas cláusulas **from** e *where* da consulta externa. Observe que o atributo *B.título* da consulta externa é usado nas consultas aninhadas para garantir que apenas os conjuntos corretos de autores e palavras-chave sejam gerados para cada título.

A SQL:2003 fornece uma variedade de operadores nos multiconjuntos, incluindo uma função **set**(*M*) que retorna uma versão livre de duplicatas de um multiconjunto *M*, uma operação agregada **intersection**, que retorna a interseção de todos os multiconjuntos em um grupo, uma operação agregada **fusion**, que retorna a união de todos os multiconjuntos em um grupo, e um predicado **submultiset**, que verifica se um multiconjunto está contido em outro multiconjunto.

O padrão SQL não fornece maneira alguma de atualizar atributos de multiconjunto, exceto atribuindo um novo valor. Por exemplo, para excluir um valor *v* de um atributo de multiconjunto *A*, precisaríamos defini-lo como (*A* **except all multiset**[*v*]).

Identidade de objeto e tipos de referência na SQL

As linguagens orientadas a objeto fornecem a capacidade de se referir a objetos. Um atributo de um tipo pode ser uma referência a um objeto de um tipo especificado. Por exemplo, na SQL, podemos definir um tipo *Departamento* com um campo *nome* e um campo *chefe* que seja uma referência ao tipo *Pessoa*, e uma tabela *departamentos* do tipo *Departamento*, da seguinte forma:

```
create type Departamento(
   nome varchar(20),
   chefe ref(Pessoa) scope pessoas
)
create table departamentos of Departamento
```

Aqui, a referência é restrita às tuplas da tabela *pessoas*. Essa restrição do escopo de uma referência às tuplas de uma tabela é obrigatória na SQL, e ela faz as referências se comportarem como chaves estrangeiras.

Podemos omitir a declaração **scope** *pessoas* da declaração de tipo e, em vez dela, fazer uma adição à instrução **create table**:

```
create table departamentos of Departamento
   (chefe with options scope pessoas)
```

A tabela referenciada precisa ter um atributo que armazena o identificador da tupla. Declaramos esse atributo, chamado o **atributo auto-referencial**, acrescentando uma cláusula **ref is** à instrução **create table**:

```
create table pessoas of Pessoa
ref is id_pessoa system generated
```

Aqui, *id_pessoa* é um nome de atributo, não uma palavra-chave, e a instrução **create table** especifica que o identificador é gerado automaticamente pelo banco de dados.

Para inicializar um atributo de referência, precisamos obter o identificador da tupla que deve ser referenciada. Podemos obter o valor do identificador de uma tupla por meio de uma consulta. Portanto, para criar uma tupla com o valor de referência, podemos primeiro criar a tupla com uma referência nula e, depois, a referência separadamente:

```
insert into departamentos
   values('CS', nulo)
update departamentos
   set chefe = (select p.id_pessoa
                from pessoas as p
                where nome = 'John')
where nome = 'CS'
```

Uma alternativa para os identificadores gerados pelo sistema é permitir que os usuários gerem identificadores. O tipo do atributo auto-referencial precisa ser especificado

como parte da definição de tipo da tabela referenciada, e a definição de tabela precisa especificar que a referência é gerada pelo usuário (**user generated**):

```
create type Pessoa
  (nome varchar(20),
  endereço varchar(20))
  ref using varchar(20)
create table pessoas of Pessoa
  ref is id_pessoa user generated
```

Ao inserir uma tupla em *pessoas*, precisamos fornecer um valor para o identificador:

```
insert into pessoas (id_pessoa, nome, endereço) values
  ('01284567', 'John', '23 Coyote Run')
```

Nenhuma outra tupla para *pessoas* ou suas supertabelas ou subtabelas pode ter o mesmo identificador. Podemos, então, usar o valor do identificador quando inserirmos uma tupla em *departamentos*, sem a necessidade de uma consulta separada para o identificador:

```
insert into departamentos
  values ('01284567')
```

É possível até mesmo usar um valor de chave primária existente como o identificador, incluindo a cláusula **ref from** na definição de tipo:

```
create type Pessoa
  (nome varchar(20) primary key,
  endereço varchar(20))
  ref from(nome)
create table pessoas of Pessoa
  ref is id_pessoa derived
```

Note que a definição de tabela precisa especificar que a referência é derivada, e precisa especificar ainda um nome de atributo auto-referencial. Ao inserir uma tupla para *departamentos*, podemos então usar

```
insert into departamentos
  values ('CS', 'John')
```

As referências são desfeitas na SQL:1999 pelo símbolo ->. Considere a tabela *departamentos* definida anteriormente. Podemos usar essa consulta para encontrar os nomes e endereços dos chefes de todos os departamentos:

```
select chefe->nome, chefe->endereço
from departamentos
```

Uma expressão como *chefe->nome* é chamada uma **expressão de caminho**.

Como *chefe* é uma referência a uma tupla na tabela *pessoas*, o atributo *nome* na consulta anterior é o atributo *nome* da tupla da tabela *pessoas*. As referências podem ser usadas para ocultar operações de junção; no exemplo anterior, sem as referências, o campo *chefe* de *departamento* seria declarado uma chave estrangeira da tabela *pessoas*. Para encontrar o nome e o endereço do chefe de um departamento, precisaríamos de uma junção explícita das relações *departamentos* e *pessoas*. O uso de referências simplifica a consulta consideravelmente.

Podemos usar a operação **deref** para retornar a tupla apontada por uma referência e, depois, acessar seus atributos, como mostrado a seguir.

```
select deref(chefe).nome
from departamentos
```

Implementando recursos O-R

Os sistemas de banco de dados relacionais de objeto são basicamente extensões dos sistemas de banco de dados relacionais existentes. As modificações são claramente necessárias em muitos níveis do sistema de banco de dados. Entretanto, para minimizar as alterações no código do sistema de armazenamento (armazenamento de relação, índices etc.), os tipos de dados complexos aceitos pelos sistemas relacionais de objeto podem ser traduzidos para o sistema de tipo mais simples dos bancos de dados relacionais.

Para entender como fazer essa tradução, precisamos apenas olhar como alguns recursos do modelo E-R são traduzidos para relações. Por exemplo, os atributos multivalores no modelo E-R correspondem aos atributos multivalores no modelo relacional de objeto. Os atributos compostos correspondem aproximadamente aos tipos estruturados. As hierarquias ISA no modelo E-R correspondem à herança de tabela no modelo relacional de objeto.

As técnicas para converter recursos do modelo E-R em tabelas, que vimos na seção "Redução aos esquemas relacionais" do Capítulo 6, podem ser usados, com algumas extensões, para traduzir dados relacionais de objeto em dados relacionais no nível de armazenamento.

As subtabelas podem ser armazenadas de uma maneira eficiente, sem a duplicação de todos os campos herdados, de uma das seguintes maneiras:

- Cada tabela armazena a chave primária (que pode ser herdada de uma tabela-pai) e os atributos são definidos localmente. Os atributos herdados (que não a chave primária) não precisam ser armazenados e podem ser derivados por meio de uma junção com as supertabelas, baseado na chave primária.

- Cada tabela armazena todos os atributos herdados e definidos localmente. Quando uma tupla é inserida, ela é armazenada apenas na tabela em que é inserida, e sua presença é suposta em cada uma das supertabelas. O acesso a todos os atributos de uma tupla é mais rápido, já que uma junção não é necessária.

  Entretanto, no caso de o sistema de tipo permitir que uma entidade seja representada em duas subtabelas sem estar presente, em uma subtabela comum de ambas, essa representação pode resultar em duplicação de informações. Além disso é difícil traduzir chaves estrangeiras referindo-se a uma supertabela para restrições nas subtabelas; para implementar eficientemente essas chaves estrangeiras, a supertabela terá de ser definida como uma visão, e o sistema de banco de dados terá de aceitar chaves estrangeiras em visões.

As implementações podem escolher representar tipos de array e multiconjunto diretamente, ou podem escolher usar uma representação normalizada internamente. As representações normalizadas costumam ocupar mais espaço e exigem um custo de junção/agrupamento extra para coletar dados em um array ou multiconjunto. Por outro lado, as representações normalizadas podem ser mais fáceis de implementar.

As interfaces de programa de aplicação ODBC e JDBC foram estendidas para recuperar e armazenar tipos estruturados; por exemplo, a JDBC fornece um método `getObject( )` que é semelhante a `getString( )` mas retorna um objeto `Struct` Java, do qual os componentes do tipo estruturado podem ser extraídos. Também é possível associar uma classe Java a um tipo estruturado SQL, e a JDBC, então, fará a conversão entre os tipos. Veja os manuais de referência da ODBC ou JDBC para obter detalhes.

Linguagens de programação persistentes

As linguagens de banco de dados diferem das linguagens de programação tradicionais, pois manipulam diretamente os dados que são persistentes – ou seja, dados que continuam a existir mesmo após o programa que os criou ter sido terminado. Uma relação em um banco de dados e tuplas em uma relação são exemplos de dados persistentes. Por outro lado, os únicos dados persistentes que as linguagens de programação tradicionais manipulam diretamente são os arquivos.

O acesso a um banco de dados é apenas um componente de qualquer aplicação do mundo real. Embora uma linguagem de manipulação de dados como SQL seja bastante eficaz para acessar dados, uma linguagem de programação é necessária para implementar outros componentes da aplicação, como interfaces com o usuário ou comunicação com outros computadores. A maneira tradicional de realizar a interface das linguagens de banco de dados com as linguagens de programação é incorporando SQL dentro da linguagem de programação.

Uma **linguagem de programação persistente** é uma linguagem de programação estendida com construções para manipular dados persistentes. As linguagens de programação persistentes podem ser distinguidas das linguagens com SQL incorporada de duas maneiras:

1. Com uma linguagem incorporada, o sistema de tipo da linguagem host normalmente difere do sistema de tipo da linguagem de manipulação de dados. O programador é responsável por qualquer tipo de conversão entre a linguagem host e a SQL. Fazer o programador realizar essa tarefa possui várias desvantagens:
   - O código para converter entre objetos e tuplas opera fora do sistema de tipo orientado a objeto e, portanto, tem uma chance maior de ter erros não detectados.
   - A conversão entre o formato orientado a objeto e o formato relacional de tuplas no banco de dados exige uma quantidade substancial de código. O código de tradução do formato, juntamente com o código para carregar e descarregar dados de um banco de dados, pode formar uma porcentagem significativa do código total necessário para uma aplicação.

   Por outro lado, em uma linguagem de programação persistente, a linguagem de consulta está totalmente integrada com a linguagem host, e ambas compartilham o mesmo sistema de tipo. Os objetos podem ser criados e armazenados no banco de dados sem qualquer tipo explícito ou mudanças de formato; quaisquer mudanças de formato necessárias são realizadas transparentemente.
2. O programador que utiliza uma linguagem de consulta incorporada é responsável por escrever código explícito para buscar dados do banco de dados para a memória. Se quaisquer atualizações forem realizadas, o programador precisa escrever código explicitamente para armazenar os dados atualizados novamente no banco de dados.

   Por outro lado, em uma linguagem de programação persistente, o programador pode manipular dados persistentes sem escrever código explicitamente a fim de transferi-lo para a memória ou armazená-lo no disco.

Nesta seção, descrevemos como as linguagens de programação orientadas a objeto, como C++ e Java, podem ser es-

tendidas para torná-las linguagens de programação persistentes. Esses recursos de linguagem permitem que os programadores manipulem dados diretamente da linguagem de programação, sem precisar utilizar uma linguagem de manipulação de dados, como SQL. Portanto, eles fornecem uma maior integração das linguagens de programação com o banco de dados do que, por exemplo, a SQL incorporada.

Existem, no entanto, certas desvantagens nas linguagens persistentes que precisamos ter em mente ao decidir se devemos ou não usá-las. Como a linguagem de programação normalmente é uma linguagem poderosa, é relativamente fácil cometer erros de programação que danifiquem o banco de dados. A complexidade da linguagem dificulta a otimização de alto nível, como a redução de E/S. O suporte para consulta declarativa é importante para muitas aplicações, mas as linguagens de programação persistentes atualmente não aceitam bem as consultas declarativas.

Neste capítulo, descrevemos uma série de aspectos que precisam ser tratados ao acrescentar persistência a uma linguagem de programação existente. Primeiro, cuidaremos das questões independentes de linguagem e, nas seções subseqüentes, veremos os aspectos que são específicos à linguagem C++ e à linguagem Java. Entretanto, não abordamos detalhes das extensões de linguagem; embora vários padrões tenham sido propostos, nenhum obteve aceitação universal. Veja as referências nas notas bibliográficas para aprender mais sobre extensões de linguagem específicas e para ver mais detalhes das implementações.

Persistência de objetos

As linguagens de programação orientadas a objeto já têm um conceito de objetos, um sistema de tipo para definir tipos de objeto e construções para criar objetos. Entretanto, esses objetos são *transientes* – eles são eliminados quando o programa termina, exatamente como as variáveis em um programa Java ou C são excluídas quando o programa termina. Se desejarmos torná-la uma linguagem de programação de banco de dados, o primeiro passo é fornecer uma maneira de tornar os objetos persistentes. Vários métodos foram propostos.

- **Persistência por classe.** A maneira mais simples, mas menos conveniente, é declarar que uma classe é persistente. Todos os objetos da classe, então, são objetos persistentes por padrão. Os objetos de classes não persistentes são todos transientes.

  Este método não é flexível, já que normalmente é útil ter tanto objetos transientes quanto persistentes em uma única classe. Muitos sistemas de banco de dados orientados a objeto interpretam a declaração de uma classe como persistente como sendo o mesmo que dizer que os objetos na classe *podem se tornar* persistentes, em vez de que todos os objetos na classe *são* persistentes. Essas classes poderiam ser chamadas, mais apropriadamente, de classes "persistíveis".
- **Persistência por criação.** Neste método, nova sintaxe é introduzida para criar objetos persistentes, estendendo a sintaxe para criar objetos transientes. Portanto, um objeto é persistente ou transiente, dependendo de como ele foi criado. Vários sistemas de banco de dados orientados a objeto seguem este método.
- **Persistência por marcação.** Uma variante do método anterior é marcar os objetos como persistentes após terem sido criados. Todos os objetos são criados como transientes, mas, se um objeto deve persistir além da execução do programa, ele precisa ser marcado explicitamente como persistente antes que o programa termine. Este método, diferente do método anterior, adia a decisão sobre persistência ou transiência para após o objeto ser criado.
- **Persistência por acessibilidade.** Um ou mais objetos são explicitamente declarados como objetos persistentes (raiz). Todos os outros objetos são persistentes se (e apenas se) eles forem acessíveis do objeto raiz por de uma seqüência de uma ou mais referências.

  Portanto, todos os objetos referenciados pelos objetos persistentes raiz (ou seja, todos os objetos cujos identificadores de objeto são armazenados nos objetos persistentes raiz) são persistentes. No entanto, também, todos os objetos referenciados a partir desses objetos são persistentes, assim como os objetos aos quais eles se referem, e assim por diante.

  Uma vantagem desse esquema é que ele facilita tornar persistentes estruturas de dados inteiras simplesmente declarando a raiz dessas estruturas como persistente. Entretanto, o sistema de banco de dados fica com a responsabilidade de seguir cadeias de referências para detectar quais objetos são persistentes, e isso pode ter um custo significativo.

Identidade de objeto e ponteiros

Em uma linguagem de programação orientada a objeto que não tenha sido estendida para manipular persistência, quando um objeto é criado, o sistema retorna um identificador de objeto transiente. Os identificadores de objeto transientes são válidos somente quando o programa que os criou está sendo executado; depois que o programa termina, os objetos são excluídos e o identificador se torna inútil. Quando um objeto persistente é criado, a ele é atribuído um identificador de objeto persistente.

A noção de identidade de objeto possui uma relação interessante com os ponteiros nas linguagens de programa-

ção. Uma maneira simples de obter identidade embutida é pelos ponteiros para locais físicos no armazenamento. Em especial, em muitas linguagens orientadas a objeto, como C++, um identificador de objeto transiente é, na verdade, um ponteiro na memória.

Entretanto, a associação de um objeto com um local físico no armazenamento pode mudar com o tempo. Existem vários graus de permanência de identidade:

- **Intraprocedimento**. A identidade persiste apenas durante a execução de um único procedimento. Exemplos de identidade intraprocedimento são as variáveis locais dentro dos procedimentos.
- **Intraprograma**. A identidade persiste apenas durante a execução de um único programa ou consulta. Exemplos de identidade intraprograma são as variáveis globais nas linguagens de programação. Os ponteiros da memória principal ou memória virtual oferecem apenas identidade intraprograma.
- **Interprograma**. A identidade persiste de uma execução de programa para outra. Os ponteiros para dados do sistema de arquivos no disco oferecem identidade interprograma, mas eles podem mudar se o modo como os dados são armazenados no sistema de arquivos for alterado.
- **Persistente**. A identidade persiste não só entre execuções de programa, mas também entre reorganizações estruturais dos dados. É a forma persistente de identidade que é necessária para os sistemas orientados a objeto.

Nas extensões persistentes de linguagens como C++, os identificadores de objeto para objetos persistentes são implementados como "ponteiros persistentes". Um *ponteiro persistente* é um tipo de ponteiro que, diferentemente dos ponteiros na memória, permanece válido mesmo após o término de uma execução de programa, e em algumas formas de reorganização de dados. Um programador pode usar um ponteiro persistente das mesmas maneiras que pode usar um ponteiro na memória em uma linguagem de programação. Conceitualmente, podemos pensar em um ponteiro persistente como um ponteiro para um objeto no banco de dados.

Armazenamento e acesso dos objetos persistentes

O que significa armazenar um objeto em um banco de dados? Evidentemente, a parte dos dados de um objeto precisa ser armazenada individualmente para cada objeto. Logicamente, o código que implementa métodos de uma classe deve ser armazenado no banco de dados como parte do seu esquema, juntamente com as definições de tipo das classes. Entretanto, muitas implementações simplesmente armazenam o código em arquivos fora do banco de dados para evitar a necessidade de integrar software de sistema, como compiladores, com o sistema de banco de dados.

Existem várias maneiras de encontrar objetos no banco de dados. Uma delas é dar nomes aos objetos, exatamente como fazemos com os arquivos. Esse método funciona para um número relativamente pequeno de objetos e não é apropriado para milhões. Uma segunda maneira é expor identificadores de objeto ou ponteiros persistentes aos objetos, que podem ser armazenados externamente. Diferente dos nomes, esses ponteiros não precisam ser mnemônicos e podem até mesmo ser ponteiros físicos para um banco de dados.

Uma terceira maneira é armazenar coleções de objetos e permitir que os programas interajam com as coleções para encontrar objetos necessários. As coleções de objetos podem, elas mesmas, ser modeladas como objetos de um *tipo de coleção*. Os tipos de coleção incluem conjuntos, multiconjuntos (isto é, conjuntos com possivelmente muitas ocorrências de um valor), listas e assim por diante. Um caso especial de uma coleção é uma *extensão de classe*, que é a coleção de todos os objetos pertencentes à classe. Se uma extensão de classe está presente para uma classe, então, sempre que o objeto da classe é criado, ele é inserido na extensão de classe automaticamente, e sempre que um objeto é excluído, ele é removido da extensão de classe. As extensões de classe permitem que classes sejam tratadas como relações, na medida em que podemos examinar todos os objetos na classe exatamente como podemos examinar todas as tuplas em uma relação.

A maioria dos sistemas de banco de dados orientados a objeto suporta todas as três formas de acessar objetos persistentes. Eles atribuem identificadores a todos os objetos. Normalmente, dão nomes apenas às extensões de classe e a outros objetos de coleção, e, talvez, a outros objetos selecionados, mas não à maioria dos objetos. Eles geralmente mantêm extensões de classe para todas as classes que podem ter objetos persistentes, mas, em muitas implementações, as extensões de classe contêm apenas objetos persistentes da classe.

Sistemas C++ persistentes

Vários bancos de dados orientados a objeto baseados em extensões persistentes à C++ apareceram nas duas últimas décadas (veja as notas bibliográficas). Existem diferenças entre eles em termos da arquitetura de sistema, embora tenham muitos recursos comuns em termos da linguagem de programação.

Vários dos recursos orientados a objeto da linguagem C++ ajudam a fornecer uma boa quantidade de suporte para a persistência sem mudar a linguagem propriamente dita. Por exemplo, podemos declarar uma classe chamada

`Persistent_Object` com atributos e métodos para dar suporte à persistência; qualquer outra classe que deveria ser persistente pode se tornar uma subclasse dessa classe e, desse modo, herdar o suporte para a persistência. A linguagem C++ (como algumas outras linguagens de programação modernas) também permite redefinir nomes de função e operadores-padrão – tal como +, –, o operador de desreferência de ponteiro –>, e assim por diante – de acordo com o tipo dos operandos em que eles são aplicados. Essa capacidade é chamada *sobrecarga* e é usada para redefinir os operadores para se comportarem da maneira necessária quando estão operando em objetos persistentes.

Fornecer suporte à persistência por meio de bibliotecas de classe tem a vantagem de fazer apenas mudanças mínimas necessárias na C++; além disso, ela é relativamente fácil de implementar. Entretanto, sua desvantagem é que o programador precisa gastar muito mais tempo para escrever um programa que manipule objetos persistentes, e não é fácil para o programador especificar restrições de integridade no esquema ou fornecer suporte para consulta declarativa. Algumas implementações C++ persistentes aceitam extensões da sintaxe C++ para tornar essas tarefas mais fáceis.

Os seguintes aspectos precisam ser observados ao incluir suporte à persistência na C++ (e em outras linguagens):

- **Ponteiros persistentes:** um novo tipo de dados precisa ser definido para representar ponteiros persistentes. Por exemplo, o padrão C++ ODMG define uma classe de modelo `d_Ref< T >` para representar ponteiros persistentes para uma classe *T*. O operador de desreferência nessa classe é redefinido para buscar o objeto no disco (se ainda não estiver presente na memória), e retorna um ponteiro na memória para o buffer em que o objeto foi buscado. Portanto, se *p* é um ponteiro persistente para uma classe *T*, pode-se usar sintaxe padrão, como `p->A` ou `p->f(v)` para acessar o atributo *A* da classe *T* ou chamar o método *f* da classe *T*.

  O sistema de banco de dados ObjectStore usa um método diferente para os ponteiros persistentes. Ele usa tipos de ponteiro normais para armazenar ponteiros persistentes. Isso apresenta dois problemas: (1) os tamanhos de ponteiro na memória podem ser de apenas 4 bytes, que é muito pequeno para usar com bancos de dados maiores que 4 gigabytes, e (2) quando um objeto é movido no disco, os ponteiros na memória para seu antigo local físico se tornam inúteis. O ObjectStore usa uma técnica chamada "swizzling de hardware" para resolver os dois problemas; ele realiza uma pré-busca dos objetos do banco de dados na memória e substitui os ponteiros persistentes por ponteiros na memória, e, quando os dados são armazenados de volta no disco, os ponteiros na memória são substituídos por ponteiros persistentes. Uma vez no disco, o valor armazenado no campo do ponteiro na memória não é o ponteiro persistente real; em vez disso, o valor é consultado em uma tabela para encontrar o valor do ponteiro persistente real.
- **Criação de objetos persistentes:** o operador `new` da C++ é usado para criar objetos persistentes definindo uma versão "sobrecarregada" do operador, que toma argumentos extras especificando que ele deve ser criado no banco de dados. Assim, em vez de `new T( )`, chamaríamos `new (db) T( )` para criar um objeto persistente, onde `db` identifica o banco de dados.
- **Extensões de classe:** as extensões de classe são criadas e mantidas automaticamente para cada classe. O padrão C++ ODMG exige que o nome da classe seja passado como um parâmetro adicional para a operação `new`. Isso também permite que múltiplas extensões sejam mantidas para uma classe, passando nomes diferentes.
- **Relações:** as relações entre classes normalmente são representadas armazenado ponteiros de cada objeto nos objetos a que estão relacionados. Objetos relacionados com múltiplos objetos de uma determinada classe armazenariam um conjunto de ponteiros. Portanto, se um par de objetos está em uma relação, cada um deve armazenar um ponteiro para o outro. Os sistemas C++ persistentes fornecem uma maneira de especificar essas restrições de integridade e de as impor criando e excluindo ponteiros automaticamente. Portanto, se um ponteiro é criado de um objeto *a* para um objeto *b*, um ponteiro para *a* é acrescentado automaticamente para o objeto *b*.
- **Interface de iteração:** como os programas precisam iterar sobre os membros de classe, uma interface é necessária para iterar sobre membros de uma extensão de classe. A interface de iteração também permite que seleções sejam especificadas, de modo que apenas os objetos que satisfazem o predicado de seleção precisem ser buscados.
- **Transações:** os sistemas C++ persistentes fornecem suporte para iniciar uma transação e para confirmá-la ou revertê-la.
- **Atualizações:** um dos objetivos de fornecer suporte a persistência para uma linguagem de programação é permitir a persistência transparente. Ou seja, uma função que opera em um objeto não deve precisar saber que o objeto é persistente; assim, as mesmas funções podem ser usadas nos objetos, quer sejam persistentes ou não.

  Entretanto, um problema resultante é que é difícil detectar quando um objeto foi atualizado. Algumas extensões persistentes da C++ exigiam que o programador especificasse explicitamente que um objeto foi modificado chamando uma função `mark_modified( )`. Além de aumentar o trabalho do programador, esse método aumenta a chance de erros de programação, resultando em um banco de dados danificado. Se um programador omitisse

uma chamada de `mark_modified( )`, possivelmente uma atualização feita por uma transação nunca teria sido propagada para o banco de dados, enquanto outra atualização feita pela mesma transação seria propagada, violando o princípio da atomicidade das transações.

Outros sistemas, como ObjectStore, usam suporte de proteção de memória fornecido pelo sistema operacional/hardware para detectar escritas em um bloco da memória e marcá-lo como um bloco sujo que deve ser escrito posteriormente no disco.

- **Linguagem de consulta:** as interfaces de iteração fornecem suporte para consultas de seleção simples. Para aceitar consultas mais complexas, os sistemas C++ persistentes definem uma linguagem de consulta.

Um grande número de sistemas de banco de dados orientados a objeto baseados na C++ foram desenvolvidos no final da década de 1980 e princípio da década de 1990. Entretanto, o mercado para esses bancos de dados foi muito menor do que o previsto, já que a maioria das necessidades de aplicação é mais do que satisfeita usando SQL por interfaces como ODBC ou JDBC. Como resultado, a maior parte dos sistemas de banco de dados orientados a objeto desenvolvidos nesse período simplesmente não existe mais. Na década de 1990, o Object Data Management Group (ODMG) definiu padrões para acrescentar persistência a C++ e Java. Entretanto, o grupo encerrou suas atividades por volta de 2002. O ObjectStore e o Versant estão entre os sistemas de banco de dados orientados a objeto originais que ainda estão disponíveis.

Embora os sistemas de banco de dados orientados a objeto não tenham alcançado o sucesso comercial que esperavam, a motivação para acrescentar persistência à linguagem de programação ainda permanece. Há várias aplicações com altas necessidades de desempenho que rodam em sistemas de banco de dados orientados a objeto; o uso da SQL impõe um overhead de desempenho muito alto para muitos desses sistemas. Com os sistemas de banco de dados relacionais de objeto agora fornecendo suporte para tipos de dados complexos, incluindo referências, está mais fácil armazenar objetos de linguagem de programação em um banco de dados SQL. Uma nova geração de sistemas de banco de dados orientados a objeto usando bancos de dados relacionais de objeto como back-end ainda pode surgir.

Sistemas Java persistentes

O uso da linguagem Java experimentou um enorme crescimento nos últimos anos. A demanda por suporte a persistência de dados nos programas Java cresceu na mesma proporção. As tentativas iniciais de criar um padrão para persistência em Java foram lideradas pelo consórcio ODMG; o consórcio encerrou seus trabalhos mais tarde, mas transferiu seu projeto para a iniciativa **Java Database Objects (JDO)**, que é coordenada pela Sun Microsystems.

O modelo JDO para persistência de objetos em programas Java difere do modelo para suporte de persistência em programas C++. Entre seus recursos estão:

- **Persistência por acessibilidade:** os objetos não são explicitamente criados em um banco de dados. Registrar explicitamente um objeto como persistente (usando o método `makePersistent( )` da classe `PersistenceManager`) torna o objeto persistente. Além disso, qualquer objeto acessível a partir de um objeto persistente se torna persistente.
- **Melhoria do código de byte:** em vez de declarar uma classe para ser persistente no código Java, as classes cujos objetos podem se tornar persistentes são especificadas em um arquivo de configuração (com sufixo `.jdo`). Um programa *enhancer* específico da implementação é executado, que lê o arquivo de configuração e realiza duas tarefas. Primeiro, ele pode criar estruturas em um banco de dados para armazenar objetos da classe. Segundo, ele modifica o código de byte (gerado pela compilação do programa Java) para manipular tarefas relacionadas à persistência. Aqui estão alguns exemplos dessas modificações:
  - Qualquer código que acessa um objeto pode ser modificado para primeiro verificar se o objeto está na memória e, se não estiver, tomar as ações para trazê-lo para a memória.
  - Qualquer código que modifica um objeto é modificado para registrar adicionalmente o objeto como modificado e, talvez, para salvar um valor pré-atualizado usado no caso de a atualização precisar ser desfeita (ou seja, se a transação for revertida).

  Outras modificações no código de byte também podem ser realizadas. Essas modificações no código de byte são possíveis desde que o código de byte seja padrão entre todas as plataformas e inclua muito mais informações do que o código de objeto compilado.
- **Mapeamento de banco de dados:** o JDO não define como os dados são armazenados no banco de dados de back-end. Por exemplo, um cenário comum é armazenar objetos em um banco de dados relacional. O programa enhancer pode criar um esquema aprimorado no banco de dados para armazenar objetos de classe. Como exatamente ele faz isso depende da implementação e não é definido pelo JDO. Alguns atributos podem ser mapeados para atributos relacionais, enquanto outros podem ser armazenados de uma forma serializada, tratados como um objeto binário pelo banco de dados. As implementações JDO podem permitir que dados relacionais

existentes sejam vistos como objetos definindo um mapeamento apropriado.

- **Extensões de classe**: as extensões de classe são criadas e mantidas automaticamente para cada classe declarada como persistente. Todos os objetos tornados persistentes são incluídos automaticamente na extensão de classe correspondente às suas classes. Os programas JDO podem acessar uma extensão de classe e iterar sobre os membros selecionados. A interface `Iterator` fornecida pela Java pode ser usada para criar repetidores em extensões de classe e para percorrer os membros da extensão de classe. O JDO também permite que seleções sejam especificadas quando um repetidor é criado em uma extensão de classe, e apenas objetos satisfazendo a seleção são buscados.
- **Tipo de referência único**: não existe diferença no tipo entre uma referência a um objeto transiente e uma referência a um objeto persistente.

Um método para conseguir essa unificação de tipos de ponteiro seria carregar o banco de dados inteiro na memória, substituindo todos os ponteiros persistentes por ponteiros na memória. Após as atualizações serem feitas, o processo seria inverso, armazenado os objetos atualizados novamente no disco. Esse método seria muito ineficiente para grandes bancos de dados.

Agora, descreveremos um método alternativo que permite que todos os objetos persistentes sejam automaticamente colocados na memória quando necessário, ao mesmo tempo que permite que todas as referências contidas nos objetos na memória sejam referências na memória. Quando um objeto A é buscado, um **objeto vazio** é criado para cada objeto B_i que ele referencia, e a cópia na memória de A possui referências ao objeto vazio correspondente para cada B_i. É claro, o sistema não tem garantia de que, se um objeto B_i já tiver sido buscado, a referência apontará para o objeto já buscado em vez de criar um novo objeto vazio. Da mesma forma, se um objeto B_i não tiver sido buscado mas tiver sido referenciado por outro objeto buscado anteriormente, ele já terá um objeto vazio criado para si; a referência ao objeto vazio existente é reutilizada, em vez de criar um novo objeto vazio.

Portanto, para cada objeto O_i que tenha sido buscado, cada referência de O_i é um objeto já buscado ou um objeto vazio. Os objetos vazios formam uma *borda* envolvendo os objetos buscados.

Sempre que o programa acessa realmente um objeto vazio O, o código de byte melhorado detecta isso e busca o objeto do banco de dados. Quando esse objeto é buscado, o mesmo processo de criar objetos vazios ocorre para todos os objetos referenciados por O. Após isso, o acesso ao objeto é permitido.[3]

Para implementar esse esquema, é necessária uma estrutura de índice na memória mapeando ponteiros persistentes para referências na memória. Ao escrever objetos novamente no disco, esse índice seria usado para substituir referências na memória por ponteiros persistentes na cópia escrita no disco.

O padrão JDO ainda está em um estágio inicial e ainda passando por revisões. Várias empresas fornecem implementações do JDO. Entretanto, ainda não se sabe se o padrão será amplamente usado, diferentemente do ODMG C++.

Orientação a objeto *versus* relacional de objeto

Neste ponto, estudamos os bancos de dados relacionais de objeto, que são bancos de dados orientados a objeto construídos sobre o modelo relacional, bem como os bancos de dados orientados a objeto, que são construídos em torno das linguagens de programação persistentes.

As extensões persistentes das linguagens de programação e os sistemas relacionais de objeto visam a mercados distintos. A natureza declarativa e a capacidade limitada (comparado a uma linguagem de programação) da linguagem SQL fornecem uma boa proteção dos dados contra erros de programação e facilita as otimizações de alto nível, como a redução de E/S. (Abordaremos a otimização das expressões relacionais no Capítulo 14.) Os sistemas relacionais de objeto visam tornar mais fáceis a modelagem de dados e as consultas utilizando tipos de dados complexos. As aplicações típicas incluem armazenamento e consulta de dados complexos, como dados de multimídia.

Entretanto, uma linguagem declarativa como a SQL impõe um ônus de desempenho significativo para certos tipos de aplicação que são executados principalmente na memória principal e que realizam um grande número de acessos ao banco de dados. As linguagens de programação persistentes visam as aplicações que possuem altas necessidades de desempenho. Elas fornecem acesso de baixo overhead aos dados persistentes e eliminam a necessidade de tradução de dados se estes precisarem ser manipulados por uma linguagem de programação. Contudo, elas são mais susce-

3. A técnica usando objetos vazios descrita aqui está intimamente relacionada com a técnica de swizzling de hardware (mencionada na seção anterior). O swizzling de hardware é usado por algumas implementações C++ para fornecer um tipo de ponteiro único para ponteiros persistentes e na memória. O swizzling de hardware usa técnicas de proteção da memória virtual fornecidas pelo sistema operacional para detectar acessos às páginas, e busca as páginas do banco de dados quando necessário. Por outro lado, a versão Java modifica o código de byte para verificar objetos vazios, em vez de usar proteção de memória, e busca objetos quando necessário, em vez de buscar páginas inteiras do banco de dados.

tíveis à danificação de dados por erros de programação e normalmente não têm uma alta capacidade de consulta. As aplicações típicas incluem os bancos de dados de CAD.

Podemos resumir as vantagens dos vários tipos de sistemas de banco de dados da seguinte maneira:

- **Sistemas relacionais**: tipos de dados simples, linguagens de consulta poderosas, alta proteção
- **Linguagem de programação persistente – baseada em OODBs**: tipos de dados complexos, integração com linguagem de programação, alto desempenho
- **Sistemas relacionais de objeto**: tipos de dados complexos, linguagens de consulta poderosas, alta proteção

Essas descrições são verdadeiras em geral, mas lembre-se de que alguns sistemas de banco de dados atravessam esses limites. Por exemplo, os sistemas de banco de dados orientados a objeto construídos em torno de uma linguagem de programação persistente podem ser implementados sobre um sistema de banco de dados relacional ou relacional de objeto. Esses sistemas podem fornecer um desempenho menor do que os sistemas de banco de dados orientados a objeto construídos diretamente em um sistema de armazenamento, mas fornecem algumas das garantias de proteção mais fortes dos sistemas relacionais.

Resumo

- O modelo de dados relacional de objeto estende o modelo de dados relacional fornecendo um sistema de tipo mais rico incluindo tipos de coleção e orientação a objeto.
- Os tipos de coleção incluem relações, conjuntos, multiconjuntos e arrays, e o modelo relacional de objeto permite que atributos de uma tabela sejam coleções.
- A orientação a objeto fornece herança com subtipos e subtabelas, bem como referências de objeto (tupla).
- O padrão SQL:1999 estende a definição de dados SQL e a linguagem de consulta para lidar com os novos tipos de dados e com orientação a objeto.
- Abordamos uma variedade de recursos da linguagem de definição de dados estendida, assim como a linguagem de consulta e, em especial, o suporte para atributos avaliados para coleção, herança e referências de tupla. Essas extensões tentam preservar as fundações relacionais – particularmente, o acesso declarativo aos dados – enquanto estende a capacidade de modelagem.
- Os sistemas de banco de dados relacionais de objeto (ou seja, os sistemas baseados no modelo de relação de objeto) fornecem um caminho de migração conveniente para os usuários de bancos de dados relacionais que desejam usar recursos orientados a objeto.
- As extensões persistentes a C++ e Java integram de modo estável e ortogonal a persistência com construções de linguagem de programação existentes e, portanto, são fáceis de usar.
- O padrão ODMG define classes e outras construções para criar e acessar objetos persistentes da C++, enquanto o padrão JDO fornece funcionalidade equivalente para Java.
- Discutimos as diferenças entre as linguagens de programação persistentes e os sistemas relacionais de objeto, e mencionamos critérios para escolher entre eles.

Termos de revisão

- Relações aninhadas
- Modelo relacional aninhado
- Tipos complexos
- Tipos de coleção
- Tipos de objeto grandes
- Conjuntos
- Arrays
- Multiconjuntos
- Tipos estruturados
- Métodos
- Tipos de linha
- Construtoras
- Herança
  - Herança única
  - Herança múltipla
- Herança de tipo
- Tipo mais específico
- Herança de tabela
- Subtabela
- Subtabelas sobrepostas
- Tipos de referência
- Escopo de uma referência
- Atributo auto-referencial
- Expressões de caminho
- Aninhamento e desaninhamento
- Funções e procedimentos SQL
- Linguagens de programação persistentes
- Persistência por
  - Classe
  - Criação
  - Marcação
  - Acessibilidade
- Associação C++ ODMG
- ObjectStore
- JDO
  - Persistência por acessibilidade
  - Raízes
  - Objetos vazios
  - Mapeamento relacional de objeto

Exercícios práticos

9.1 Uma empresa de aluguel de automóveis mantém um banco de dados para todos os veículos em sua frota atual. Para todos os veículos, ela inclui o número de identificação do veículo, o número da licença, o fabricante, o modelo, a data de compra e a cor. Dados especiais são incluídos para certos tipos de veículo:

- Caminhões: capacidade de carga
- Carros esportivos: potência do motor, idade máxima exigida para o locatário
- Vans: número de passageiros
- Veículos off-road: altura do solo, tração (duas ou quatro rodas)

Construa uma definição de esquema SQL:1999 para esse banco de dados. Use herança quando apropriado.

9.2 Considere um esquema de banco de dados com uma relação *Emp* cujos atributos são mostrados a seguir, com tipos especificados para atributos multivalores.

Emp = (*nome_emp*, *ConjuntoCrianças* **multiset**(*Crianças*), *ConjuntoHabilidades* **multiset**(*Habilidades*))
Crianças = (*nome*, *nascimento*)
Habilidades = (*tipo*, *ConjuntoExames* **setof**(*Exames*))
Exames = (*ano*, *cidade*)

a. Defina esse esquema na SQL:2003, com tipos apropriados para cada atributo.
b. Usando esse esquema, escreva as seguintes consultas na SQL:2003.
i. Encontre os nomes de todos os empregados que possuem um filho nascido em 1º de janeiro de 2000 ou depois.
ii. Encontre os empregados que fizeram um exame para o tipo de habilidade "digitação" na cidade de "Dayton".
iii. Liste todos os tipos de habilidade na relação *Emp*.

9.3 Considere o diagrama E-R na Figura 9.5, que contém atributos compostos, multivalores e derivados.
a. Dê uma definição de esquema SQL:2003 correspondente ao diagrama E-R.
b. Dê construtoras para cada um desses tipos estruturados definidos.

9.4 Considere o esquema relacional mostrado na Figura 9.6.
a. Dê uma definição de esquema na SQL:2003 correspondente ao esquema relacional, mas usando referências para expressar relações de chave estrangeira.
b. Escreva cada uma das consultas dadas no Exercício 2.9 nesse esquema, usando a SQL:2003.

9.5 Suponha que você tenha contratado um consultor para escolher um sistema de banco de dados para a aplicação do seu cliente. Para cada uma das seguintes aplicações, informe qual tipo de sistema de banco de dados (relacional, banco de dados orientado a objeto baseado em linguagem de programação persistente, relacional de objeto; não especifique um produto comercial) você recomendaria. Justifique sua recomendação.
a. Um sistema de projeto auxiliado por computador (CAD) para um fabricante de aeronaves
b. Um sistema para controlar contribuições feitas para candidatos a cargo público
c. Um sistema de informações para apoiar a produção de filmes

9.6 Em que o conceito de um objeto no modelo orientado a objeto difere do conceito de uma entidade no modelo de relação de entidade?

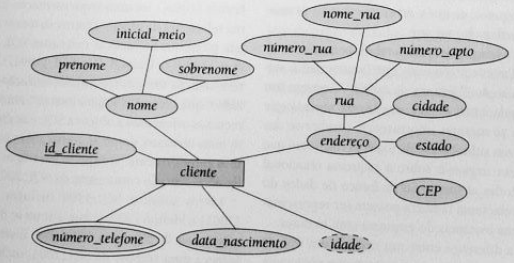


Figura 9.5 Diagrama E-R com atributos compostos, multivalores e derivados.

empregado (nome_pessoa, rua, cidade)
trabalha (nome_pessoa, nome_empresa, salário)
empresa (nome_empresa, cidade)
gerência (nome_pessoa, nome_gerente)

Figura 9.6 Banco de dados relacional para o Exercício prático 9.4.

Exercícios

9.7 Redesenhe o banco de dados do Exercício prático 9.2 na primeira forma normal (1FN) e na quarta forma normal (4FN). Liste quaisquer dependências funcionais ou multivalores que você considerar. Relacione também todas as restrições de integridade referencial que devem estar presentes nos esquemas 1FN e 4FN.

9.8 Considere o esquema do Exercício de prático 9.2.

a. Forneça instruções DDL SQL:2003 para criar uma relação *EmpA* que tenha as mesmas informações de *Emp*, mas na qual os atributos avaliados para multiconjunto *ConjuntoCrianças*, *ConjuntoHabilidades* e *ConjuntoExames* são substituídos pelos atributos avaliados para array *ArrayCrianças*, *ArrayHabilidades* e *ArrayExames*.

b. Escreva uma consulta para converter dados do esquema de *Emp* para o esquema de *EmpA*, com o array de crianças ordenado por data de nascimento, o array de habilidades ordenado pelo tipo e o array de exames ordenado pelo ano.

c. Escreva uma instrução SQL para atualizar a relação *Emp* acrescentando um filho de nome Jeb, com uma data de nascimento de 5 de fevereiro de 2001, ao empregado de nome George.

d. Escreva uma instrução SQL para realizar a mesma atualização anterior, mas na relação *EmpA*. Certifique-se de que o array de crianças permaneça ordenado por ano.

9.9 Considere os esquemas para a tabela *pessoas* e as tabelas *estudantes* e *professores*, que foram criados sob *pessoas* na seção "Herança de tabela". Forneça um esquema relacional na terceira forma normal que represente as mesmas informações. Lembre-se das restrições nas subtabelas e dê todas as restrições que precisam ser impostas sobre o esquema relacional para que todas as instâncias de banco de dados do esquema relacional também possam ser representadas por uma instância do esquema com herança.

9.10 Explique a diferença entre um tipo x e um tipo de referência **ref**(x). Sob que circunstâncias você escolheria usar um tipo de referência?

9.11 a. Dê uma definição de esquema SQL:1999 do diagrama E-R na Figura 9.7, que contém especializações, usando subtipos e subtabelas.

b. Forneça uma consulta SQL:1999 para encontrar os nomes de todas as pessoas que não são secretários.

c. Forneça uma consulta SQL:1999 para imprimir os nomes das pessoas que não são nem empregados nem clientes.

d. Você pode criar uma pessoa que seja um empregado e um cliente com o esquema que você elaborou? Explique como, ou explique por que não é possível.

9.12 Suponha que um banco de dados JDO tivesse um objeto A, que referencia o objeto B, que, por sua vez, referencia o objeto C. Considere que todos os objetos estão no disco inicialmente. Suponha que um programa primeiro desreferencia A, depois desreferencia B seguindo a referência de A, e, finalmente, desreferencia C. Mostre os objetos que são representados na memória após cada desreferência, juntamente com seu estado (vazio ou preenchido) e os valores em seus campos de referência.

Notas bibliográficas

Várias extensões orientadas a objeto para a SQL foram propostas. POSTGRES (Stonebraker e Rowe [1986] e Stonebraker [1986]) foi uma implementação inicial de um sistema relacional de objeto. Outros sistemas relacionais de objeto pioneiros incluem as extensões SQL do O_2 (Bancilhon *et al.* [1989]) e UniSQL (UniSQL [1991]). A SQL:1999 foi o resultado de um esforço de padronização extenso (e demorado), que começou originalmente como um acréscimo de recursos orientados a objeto à SQL e acabou incluindo muito mais recursos, como as construções procedurais que vimos anteriormente. O suporte para tipos de multiconjunto foi acrescentado como parte da SQL:2003.

Livros sobre a SQL:1999 incluem Melton e Simon [2001] e Melton [2002]; este último se dedica aos recursos relacionais de objeto da SQL:1999. Eisenberg *et al.* [2004] fornece uma sinopse da SQL:2003, incluindo seu suporte para multiconjuntos. Consulte os manuais (on-line) do sis-

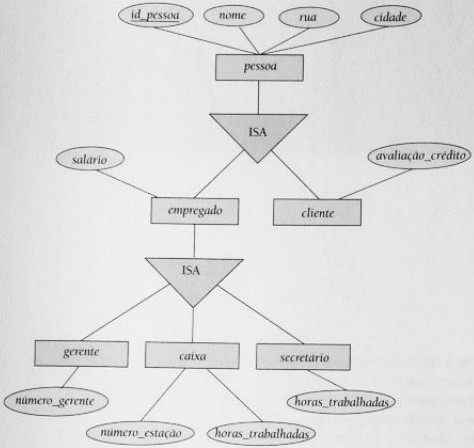


Figura 9.7 Especialização e generalização.

tema de banco de dados que você usa para descobrir que recursos da SQL:1999/SQL:2003 ele aceita.

Diversos sistemas de banco de dados orientados a objeto foram desenvolvidos no final da década de 1980 e início da década de 1990. Entre os sistemas comerciais mais notáveis estavam o ObjectStore (Lamb *et al.* [1991]), O_2 (Lecluse *et al.* [1998]) e Versant. O padrão de banco de dados de objeto ODMG é descrito em detalhes em Cattell [2000]. O JDO é descrito por Roos [2002], Tyagi *et al.* [2003] e Jordan e Russell [2003].

Ferramentas

Existem diferenças consideráveis entre os diversos produtos de banco de dados em seu suporte para recursos relacionais de objeto. O Oracle provavelmente possui o suporte mais extensivo entre os principais fornecedores de banco de dados. O sistema de banco de dados Informix fornece suporte para muitos recursos relacionais de objeto. Tanto o Oracle quanto o Informix dispunham de recursos relacionais de objeto antes que o padrão SQL:1999 fosse finalizado, e possuem muitos recursos que não são parte da SQL:1999.

Informações sobre o ObjectStore e Versant, incluindo download de versões de avaliação, podem ser obtidas de seus respectivos sites (`objectstore.com` e `versant.com`). O projeto de banco de dados Apache (`db.apache.org`) fornece uma ferramenta de mapeamento relacional de objeto para Java que aceita APIs ODMG Java e JDO. Uma implementação de referência do JDO pode ser obtida em `sun.com`; use um mecanismo de busca para obter o URL completo.

CAPÍTULO 10

XML

Diferente da maioria das tecnologias apresentadas nos capítulos anteriores, a **Extensible Markup Language** (**XML**) não foi concebida originalmente como uma tecnologia de banco de dados. Na verdade, como a *Hyper-Text Markup Language* (HTML), em que a World Wide Web é baseada, a XML tem suas raízes no gerenciamento de documentos, sendo derivada de uma linguagem para estruturar grandes documentos, conhecida como Standard Generalized Markup Language (SGML). Porém, ao contrário da SGML e HTML, XML pode representar dados de banco de dados, além de muitos outros tipos de dados estruturados. Ela é particularmente útil como um formato de dados quando uma aplicação precisa se comunicar com outra aplicação, ou integrar informações de várias outras aplicações. Quando a XML é usada nesses contextos, surgem muitas questões de banco de dados, incluindo como organizar, manipular e consultar os dados XML. Neste capítulo, apresentamos a XML e discutimos tanto o gerenciamento de dados XML com técnicas de banco de dados quanto a troca de dados formatados como documentos XML.

Motivação

Para entender a XML, é importante entender suas raízes como uma linguagem de marcação de documento. O termo **marcação** refere-se a qualquer elemento em um documento que não sirva como parte da saída impressa. Por exemplo, um escritor criando texto que por fim será composto em uma revista pode querer fazer anotações sobre como a composição deverá ser feita. Seria importante digitar essas notas de uma maneira que elas pudessem ser distinguidas do conteúdo real, de modo que uma nota como "componha esta palavra em uma fonte com tamanho grande e negrito" ou "insira uma quebra de linha aqui" não acabe sendo impressa na revista. Essas notas transmitem informações extras sobre o texto. No processamento eletrônico de documentos, uma **linguagem de marcação** é uma descrição formal de qual parte do documento é conteúdo, qual parte é marcação e o que significa a marcação.

Assim como os sistemas de banco de dados evoluíram a partir do processamento físico do arquivo, para oferecerem uma visão lógica separada, as linguagens de marcação evoluíram a partir da especificação de instruções sobre como imprimir partes do documento para especificar a *função* do conteúdo. Por exemplo, com a marcação funcional, o texto que representa cabeçalhos de seção (para esta seção, a palavra "Motivação") seria marcado como sendo um cabeçalho de seção, em vez de ser marcado como texto a ser impresso em uma fonte de tamanho grande e negrito. Do ponto de vista da composição, essa marcação funcional permite que o documento seja formatado de modo diferente em situações diferentes. Isso também ajuda diferentes partes de um documento grande, ou diferentes páginas de um site grande, a serem formatadas de uma maneira uniforme. Mais importante, a marcação funcional também ajuda a registrar o que cada parte do texto representa semanticamente e, ao mesmo tempo, ajuda a automatizar a extração das principais partes dos documentos.

Para a família de linguagens de marcação, que inclui HTML, SGML e XML, a marcação toma a forma de **tags** delimitadas em sinais < e >. As tags são usadas em pares, com <tag> e </tag> delimitando o início e o final da parte do documento à qual a tag se refere. Por exemplo, o título de um documento poderia ser marcado da seguinte forma:

```
<titulo>Conceitos de Sistemas de Banco de
Dados</titulo>
```

```
<banco>
   <conta>
        <número_conta> A-101 </número_conta>
        <nome_agência> Downtown </nome_agência>
        <saldo> 500 </saldo>
   </conta>
   <conta>
        <número_conta> A-102 </número_conta>
        <nome_agência> Perryridge </nome_agência>
        <saldo> 400 </saldo>
   </conta>
   <conta>
        <número_conta> A-201 </número_conta>
        <nome_agência> Brighton </nome_agência>
        <saldo> 900 </saldo>
   </conta>
   <cliente>
        <nome_cliente> Johnson </nome_cliente>
        <rua_cliente> Alma </rua_cliente>
        <cidade_cliente> Palo Alto </cidade_cliente>
   </cliente>
   <cliente>
        <nome_cliente> Hayes </nome_cliente>
        <rua_cliente> Main </rua_cliente>
        <cidade_cliente> Harrison </cidade_cliente>
   </cliente>
   <depositante>
        <número_conta> A-101 </número_conta>
        <nome_cliente> Johnson </nome_cliente>
   </depositante>
   <depositante>
        <número_conta> A-201 </número_conta>
        <nome_cliente> Johnson </nome_cliente>
   </depositante>
   <depositante>
        <número_conta> A-102 </número_conta>
        <nome_cliente> Hayes </nome_cliente>
   </depositante>
</banco>
```

Figura 10.1 Representação em XML de informações bancárias.

Ao contrário da HTML, XML não prescreve o conjunto de tags permitidas, e o conjunto pode ser escolhido conforme a necessidade por cada aplicação. Esse recurso é a chave para a principal função da XML na representação e troca de dados, enquanto HTML é usada principalmente para a formatação de documentos.

Por exemplo, em nossa aplicação bancária, as informações de conta e cliente podem ser representadas como parte de um documento XML, como na Figura 10.1. Observe o uso de tags como `conta` e `número_conta`. Essas tags oferecem contexto para cada valor e permitem que a semântica do valor seja identificada. Para este exemplo, a representação de dados em XML não oferece qualquer benefício significativo em relação à representação de dados relacional tradicional; porém, usamos esse exemplo como nosso exemplo corrente, devido à sua simplicidade.

A Figura 10.2, que mostra como a informação sobre uma ordem de compra pode ser representada em XML, ilustra um uso mais realista da XML. As ordens de compra normalmente são geradas por uma organização e enviadas a outra. Tradicionalmente, elas eram impressas em papel pelo comprador e enviadas ao fornecedor; os dados seriam redigitados manualmente em um sistema de computador pelo fornecedor. Esse processo lento pode ser bastante agilizado pelo envio de informações eletronicamente entre o comprador e o fornecedor. A representação aninhada permite que todas as informações em uma ordem de compra sejam representadas naturalmente em um único documento. (As ordens de compra reais possuem muito mais informações do que representamos neste exemplo simplificado.) A XML oferece um modo padrão de marcar os dados; naturalmente, as duas organizações precisam combinar so-

```
<ordem_compra>
   <identificador> P-101 </identificador>
   <comprador>
      <nome> Crazy Coyote </nome>
      <endereço> Mesa Flat, Route 66, Arizona 12345, USA </endereço>
   </comprador>
   <fornecedor>
      <nome> Acme Supplies </nome>
      <endereço> 1, Broadway, New York, NY, USA </endereço>
   </fornecedor>
   <lista_itens>
      <item>
         <identificador> RS1 </identificador>
         <descrição> Atom powered rocket sled </descrição>
         <quantidade> 2 </quantidade>
         <preço> 199.95 </preço>
      </item>
      <item>
         <identificador> SG2 </identificador>
         <descrição> Superb glue </descrição>
         <quantidade> 1 </quantidade>
         <unidade-de-medida> liter </unidade-de-medida>
         <preço> 29.95 </preço>
      </item>
   </lista_itens>>
   <custo_total> 429.85 </custo_total>
   <termos_pagamento> Cash-on-delivery </termos_pagamento>
   <modo_compra> 1-second-delivery </modo_compra>
</ordem_compra>
```

Figura 10.2 Representação em XML de uma ordem de compra.

bre quais tags aparecem na ordem de compra e o que elas significam.

Em comparação com o armazenamento de dados em um banco de dados relacional, a representação XML pode ser ineficaz, pois os nomes de tag são repetidos em todo o documento. Porém, apesar dessa desvantagem, uma representação XML possui vantagens significativas quando utilizada para trocar dados entre organizações, e para armazenar informações estruturadas em arquivos:

- Primeiro, a presença das tags torna a mensagem **autodocumentável**; ou seja, um esquema não precisa ser consultado para se entender o significado do texto. Podemos prontamente ler o fragmento anterior, por exemplo.
- Segundo, o formato do documento não é rígido. Por exemplo, se alguém acrescentar informações adicionais, como uma tag `último_acesso`, indicando a última data em que a conta foi acessada, o destinatário dos dados XML pode simplesmente ignorar a tag. Como outro exemplo, na Figura 10.2, o item com identificador SG2 possui uma tag chamada `unidade-de-medida` sendo especificada, que o primeiro item não possui. A tag é exigida para itens que são encomendados por peso ou volume, e pode ser omitida para itens que são encomendados simplesmente por número.

  A capacidade de reconhecer e ignorar tags inesperadas permite que o formato dos dados evolua com o passar do tempo, sem invalidar as aplicações existentes. De modo semelhante, a capacidade de ter várias ocorrências da mesma tag facilita a representação de atributos com múltiplos valores.
- Terceiro, a XML permite estruturas aninhadas. A ordem de compra mostrada na Figura 10.2 ilustra os benefícios de ter uma estrutura aninhada. Cada ordem de compra possui um comprador e uma lista de itens como duas de suas estruturas aninhadas. Cada item, por sua vez, possui um identificador de item, descrição e um preço, aninhados dentro dele, enquanto o comprador possui nome e endereço aninhados dentro dele.

  Essa informação teria sido dividida em múltiplas relações em um esquema relacional. A informação do item teria sido armazenada em uma relação, a informação do comprador em uma segunda relação, as ordens de compra em uma terceira, e o relacionamento entre ordens de compra, compradores e itens teria sido armazenado em uma quarta relação.

```
...
   <conta>
      Esta conta agora raramente é utilizada.
      <número_conta> A-102 </número_conta>
      <nome_agência> Perryridge </nome_agência>
      <saldo> 400 </saldo>
   </conta>
...
```

Figura 10.3 Mistura de texto com subelementos.

A representação relacional ajuda a evitar redundância; por exemplo, as descrições de item seriam armazenadas apenas uma vez para cada identificador de item em um esquema relacional normalizado. Porém, na ordem de compra XML, as descrições podem ser repetidas em várias ordens de compra que pedem o mesmo item. Entretanto, reunir toda a informação relacionada a uma ordem de compra em uma única estrutura aninhada, mesmo com prejuízo da redundância, é atraente quando a informação deve ser trocada com entidades externas.

- Finalmente, como o formato XML é bastante aceito, existe uma grande variedade de ferramentas disponível para auxiliar no seu processamento, inclusive APIs de linguagem de programação para criar e ler dados XML, software de navegador e ferramentas de banco de dados.

Vamos descrever várias aplicações para dados XML mais adiante, na seção "Aplicações XML". Assim como SQL é a *linguagem* dominante para a consulta de dados relacionais, XML tornou-se o *formato* dominante para a troca de dados.

Estrutura de dados XML

A construção fundamental em um documento XML é o **elemento**. Um elemento é simplesmente um par de tags correspondentes de início e fim, com todo o texto que aparece entre elas.

Os documentos XML precisam ter um único elemento **raiz**, que compreende todos os outros elementos no documento. No exemplo da Figura 10.1, o elemento <banco> forma o elemento raiz. Além disso, os elementos em um documento XML precisam **aninhar** corretamente. Por exemplo,

```
<conta> ... <sado> ... </saldo> ... </conta>
```

está aninhado corretamente, enquanto

```
<conta> ... <saldo> ... </conta> ... </saldo>
```

não está aninhado corretamente.

Embora o aninhamento apropriado seja uma propriedade de intuitiva, podemos defini-lo mais formalmente. Diz-se que o texto aparece **no contexto de** um elemento se ele aparecer entre a tag de início e a tag de fim desse elemento. A tags são aninhadas corretamente se cada tag de início tiver uma única tag de fim correspondente que esteja no contexto do mesmo elemento pai.

Observe que o texto pode ser misturado com os subelementos de um elemento, como na Figura 10.3. Assim como em vários outros recursos da XML, essa liberdade faz mais sentido em um contexto de processamento de documento do que em um contexto de processamento de dados, e não é particularmente útil para representar dados mais estruturados, como o conteúdo do banco de dados, em XML.

A capacidade de aninhar elementos dentro de outros elementos oferece um modo alternativo de representar informações. A Figura 10.4 mostra uma representação da informação bancária da Figura 10.1, mas com os elementos `conta` aninhados dentro dos elementos `cliente`. A representação aninhada facilita a localização de todas as contas de um cliente, embora armazene os elementos `conta` de forma redundante, caso pertençam a vários clientes.

As representações aninhadas são muito usadas em aplicações de intercâmbio de dados XML para evitar uniões. Por exemplo, uma ordem de compra armazenaria o endereço completo do emissor e receptor de forma redundante em várias ordens de compra, enquanto uma representação normalizada pode exigir uma união de registros de ordem de compra com uma relação *empresa_endereço* para obter a informação de endereço.

Além dos elementos, a XML especifica a noção de um **atributo**. Por exemplo, o tipo de uma conta pode ser representado como um atributo, como na Figura 10.5. Os atributos de um elemento aparecem como pares *nome=valor*, antes do ">" de fechamento de uma tag. Os atributos são strings, e não contêm marcação. Além do mais, os atributos só podem aparecer uma vez em determinada tag, ao contrário dos subelementos, que podem ser repetidos.

Observe que, em um contexto de construção de documento, a distinção entre subelemento e atributo é importante – um atributo é implicitamente o texto que não aparece no documento impresso ou exibido. Porém, em aplicações de banco de dados e troca de dados da XML, essa distinção é menos relevante, e a escolha de representar dados

```
<banco-1>
  <cliente>
      <nome_cliente> Johnson </nome_cliente>
      <rua_cliente> Alma </rua_cliente>
      <cidade_cliente> Palo Alto </cidade_cliente>
      <conta>
        <número_conta> A-101 </número_conta>
        <nome_agência> Downtown </nome_agência>
        <saldo> 500 </saldo>
      </conta>
      <conta>
        <número_conta> A-201 </número_conta>
        <nome_agência> Brighton </nome_agência>
        <saldo> 900 </saldo>
      </conta>
  </cliente>
  <cliente>
      <nome_cliente> Hayes </nome_cliente>
      <rua_cliente> Main </rua_cliente>
      <cidade_cliente> Harrison </cidade_cliente>
      <conta>
        <número_conta> A-102 </número_conta>
        <nome_agência> Perryridge </nome_agência>
        <saldo> 400 </saldo>
      </conta>
  </cliente>
</banco-1>
```

Figura 10.4 Representação XML aninhada das informações bancárias.

```
...
   <conta tipo_conta= "corrente">
      <número_conta> A-102 </número_conta>
      <nome_agência> Perryridge </nome_agência>
      <saldo> 400 </saldo>
   </conta>
...
```

Figura 10.5 Uso de atributos.

como um atributo ou como um subelemento normalmente é arbitrária.

Uma última observação sintática é que um elemento na forma <elemento></elemento>, que não contém subelementos ou texto, pode ser abreviada como <elemento/>; porém, os elementos abreviados podem conter atributos.

Como os documentos XML são projetados para a troca entre aplicações, um mecanismo de **namespace** (espaço de nome) foi introduzido para permitir que as organizações especifiquem nomes globalmente exclusivos sendo usados como tags de elemento nos documentos. A idéia de um namespace é iniciar cada tag ou atributo com um identificador universal de recurso (por exemplo, um endereço Web). Assim, por exemplo, se o First Bank quisesse garantir que os documentos XML que ele criou não dupliquem tags usadas pelos documentos XML de qualquer parceiro de negócios, ele pode iniciar um identificador exclusivo usando um sinal de dois-pontos antes de cada nome de tag. O banco pode usar um URL da Web, por exemplo,

```
http://www.FirstBank.com
```

como um identificador exclusivo. O uso de identificadores exclusivos longos em cada tag seria um tanto inconveniente, de modo que o padrão de namespace oferece um modo de definir uma abreviação para os identificadores.

Na Figura 10.6, o elemento raiz (`banco`) tem um atributo `xmlns:FB`, que declara que `FB` é definido como uma abreviação para o URL indicado. A abreviação pode, então, ser usada em várias tags de elemento, conforme ilustramos na figura.

Um documento pode ter mais de um namespace, declarados como parte do elemento raiz. Diferentes elementos podem, então, ser associados a diferentes namespaces. Um

```
<banco xmlns:FB="http://www.FirstBank.com">
  ...
  <FB:agência>
    <FB:nome_agência> Downtown </FB:nome_agência>
    <FB:cidade_agência> Brooklyn </FB:cidade_agência>
  </FB:agência>
  ...
</banco>
```

Figura 10.6 Nomes de tag exclusivos podem ser atribuídos por meio de namespaces.

namespace padrão pode ser definido com o uso do atributo `xmlns` em vez de `xmlns:FB` no elemento raiz. Então, os elementos sem um prefixo de namespace explícito pertenceriam ao namespace padrão.

Às vezes, precisamos armazenar valores contendo tags sem que as tags sejam interpretadas como tags XML. Para podermos fazer isso, a XML permite esta construção:

```
<![CDATA[<conta> ...</conta>]]>
```

Por estar delimitada com `CDATA`, o texto `<conta>` é tratado como dados de texto normais, e não como uma tag. O termo `CDATA` significa "character data" (dados de caractere).

Esquema do documento XML

Os bancos de dados possuem esquemas, que são usados para restringir quais informações podem ser armazenadas no banco de dados e restringir os tipos de dados das informações armazenadas. Ao contrário, por padrão, os documentos XML podem ser criados sem qualquer esquema associado: um elemento pode, então, ter qualquer subelemento ou atributo. Embora tal liberdade possa ocasionalmente ser aceitável, dada a natureza de autodescrição do formato dos dados, ela geralmente não é útil quando os documentos XML tiverem de ser processados automaticamente como parte de uma aplicação, ou mesmo quando grandes quantidades de dados relacionados devem ser formatadas em XML.

Aqui, descrevemos a primeira linguagem de definição de esquema incluída como parte do padrão XML, a *Document Type Definition*, além do seu substituto definido mais recentemente, a *XMLSchema*. Outra linguagem de definição de esquema XML, chamada Relax NG, também está em uso, mas não vamos explicá-la aqui; para obter mais informações sobre Relax NG, consulte as referências na seção de notas bibliográficas.

Document Type Definition

A Document Type Definition (DTD) é uma parte opcional de um documento XML. A finalidade principal de uma DTD é muito parecida com a de um esquema: restringir as informações e os tipos de informações presentes no documento. Porém, a DTD na verdade não restringe os tipos no sentido dos tipos básicos, como inteiro ou string. Em vez disso, ela só restringe o surgimento de subelementos e atributos dentro de um elemento. A DTD é principalmente uma lista de regras indicando que padrão de subelementos pode aparecer dentro de um elemento. A Figura 10.7 mostra uma parte de uma DTD de exemplo para um documento de informações bancárias; o documento XML na Figura 10.1 está em conformidade com essa DTD.

Cada declaração está na forma de uma expressão regular para os subelementos de um elemento. Assim, na DTD da Figura 10.7, um elemento `banco` consiste em um ou mais ele-

```
<!DOCTYPE banco [
  <!ELEMENT banco ( (conta|cliente|depositante)+)>
  <!ELEMENT conta ( número_conta nome_agência saldo )>
  <!ELEMENT cliente ( nome_cliente rua_cliente cidade_cliente )>
  <!ELEMENT depositante ( nome_cliente número_conta )>
  <!ELEMENT número_conta ( #PCDATA )>>
  <!ELEMENT nome_agência ( #PCDATA )>
  <!ELEMENT saldo( #PCDATA )>
  <!ELEMENT nome_cliente( #PCDATA )>
  <!ELEMENT rua_cliente( #PCDATA )>
  <!ELEMENT cidade_cliente( #PCDATA )>
>
```

Figura 10.7 Exemplo de uma DTD.

```
<!DOCTYPE banco-2 [
    <!ELEMENT  conta ( agência, saldo )>
    <!ATTLIST conta
        número_conta ID #REQUIRED
        proprietários IDREFS #REQUIRED >
    <!ELEMENT cliente ( nome_cliente, rua_cliente, cidade_cliente )>
    <!ATTLIST cliente
        id_cliente ID #REQUIRED
        contas IDREFS #REQUIRED >
    ... declarações para agência, saldo, nome_cliente,
        rua_cliente e cidade_cliente ...
] >
```

Figura 10.8 DTD com os tipos de atributo `ID` e `IDREFS`.

mentos `conta`, `cliente` ou `depositante`; o operador | especifica "ou", enquanto o operador + especifica "um ou mais". Embora não mostrado aqui, o operador * é usado para especificar "zero ou mais", enquanto o operador ? é usado para especificar um elemento opcional (ou seja, "zero ou um").

O elemento `conta` é definido para conter subelementos `número_conta`, `nome_agência` e `saldo` (nessa ordem). De modo semelhante, `cliente` e `depositante` têm os atributos em seu esquema definidos como subelementos.

Finalmente, os elementos `número_conta`, `nome_agência`, `saldo`, `nome_cliente`, `rua_cliente` e `cidade_cliente` são todos declarados como sendo do tipo `#PCDATA`. A palavra-chave `#PCDATA` indica dados de texto; historicamente, ela deriva seu nome de "parsed character data" (dados de caractere analisados). As outras declarações de tipo especiais são `empty`, o que indica que o elemento não possui conteúdo, e `any`, que diz que não existe restrição sobre os subelementos do elemento; ou seja, quaisquer elementos, mesmo aqueles não mencionados na DTD, podem ocorrer como subelementos do elemento. A ausência de uma declaração para um elemento é equivalente a declarar o tipo explicitamente como `any`.

Os atributos permitidos para cada elemento também são declarados na DTD. Ao contrário dos subelementos, nenhuma ordem é imposta sobre os atributos. Os atributos podem ser especificados como sendo do tipo `CDATA`, `ID`, `IDREF` ou `IDREFS`; o tipo `CDATA` simplesmente diz que o atributo contém dados de caractere, enquanto os outros três não são tão simples; eles serão explicados com mais detalhes em breve. Por exemplo, a linha a seguir, vinda de uma DTD, especifica que o elemento `conta` possui um atributo do tipo `tipo_conta`, com o valor-padrão `corrente`.

```
<!ATTLIST conta tipo_conta CDATA "corrente" >
```

Os atributos precisam ter uma declaração de tipo e uma declaração padrão. A declaração padrão pode consistir em um valor-padrão para o atributo ou `#REQUIRED`, significando que um valor precisa ser especificado para o atributo em cada elemento, ou `#IMPLIED`, significando que nenhum valor-padrão foi fornecido, e o documento poderá omitir esse atributo. Se um atributo tiver um valor-padrão, para cada elemento que não especifica um valor para o atributo, o valor-padrão será preenchido automaticamente quando o documento XML for lido.

Um atributo do tipo `ID` oferece um identificador exclusivo para o elemento; um valor que ocorre em um atributo `ID` de um elemento não pode ocorrer em qualquer outro elemento no mesmo documento. No máximo um atributo de um elemento tem permissão para ser do tipo `ID`.

Um atributo do tipo `IDREF` é uma referência a um elemento; o atributo precisa conter um valor que aparece no atributo `ID` de algum elemento no documento. O tipo `IDREFS` permite uma lista de referências, separadas por espaços.

A Figura 10.8 mostra uma DTD de exemplo em que os relacionamentos de conta do cliente são representados pelos atributos `ID` e `IDREFS`, em vez de registros `depositante`. Os elementos `conta` utilizam `número_conta` como seu atributo identificador; para fazer isso, `número_conta` tornou-se um atributo de `conta`, em vez de um subelemento. Os elementos `cliente` possuem um novo atributo identificador, chamado `id_cliente`. Além disso, cada elemento `cliente` contém um atributo `contas`, do tipo `IDREFS`, que é uma lista de identificadores das contas que pertencem ao cliente. Cada elemento `conta` possui um atributo `proprietários`, do tipo `IDREFS`, que é uma lista dos proprietários da conta.

A Figura 10.9 mostra um documento XML de exemplo baseado na DTD da Figura 10.8. Observe que usamos um conjunto de contas e clientes diferente do nosso exemplo anterior, a fim de ilustrar melhor o recurso `IDREFS`.

Os atributos `ID` e `IDREF` têm a mesma finalidade dos mecanismos de referência dos bancos de dados orientados a objeto e objeto relacional, permitindo a construção de relacionamentos de dados complexos.

Definições de tipo de documento (DTDs) estão fortemente ligadas à herança de formatação de documento da XML. Por causa disso, elas são inadequadas de muitas ma-

```
<banco-2>
    <conta número_conta="A-401" proprietários="C100 C102">
        <nome_agência> Downtown </nome_agência>
        <saldo> 500 </saldo>
    </conta>
    <conta número_conta="A-402" proprietários="C102 C101">
        <nome_agência> Perryridge </nome_agência>
        <saldo> 900 </saldo>
    </conta>
    <cliente id_cliente="C100" contas="A-401">
        <nome_cliente>Joe</nome_cliente>
        <rua_cliente> Monroe </rua_cliente>
        <cidade_cliente> Madison </cidade_cliente>
    </cliente>
    <cliente id_cliente="C101" contas="A-402">
        <nome_cliente>Lisa</nome_cliente>
        <rua_cliente> Mountain </rua_cliente>
        <cidade_cliente> Murray Hill </cidade_cliente>
    </cliente>
    <cliente id_cliente="C102" contas="A-401 A-402">
        <nome_cliente>Mary</nome_cliente>
        <rua_cliente> Erin </rua_cliente>
        <cidade_cliente> Newark </cidade_cliente>
    </cliente>
</banco-2>
```

Figura 10.9 Dados XML com atributos `ID` e `IDREF`.

neiras para servir como estrutura de tipo da XML para aplicações de processamento de dados. Apesar disso, diversos formatos de troca de dados foram definidos em termos de DTDs, pois faziam parte do padrão original. A seguir, relacionamos algumas das limitações das DTDs como um mecanismo de esquema.

- Elementos de texto e atributos individuais não podem ter outra definição de tipo. Por exemplo, o elemento `saldo` não pode ser restringido a um número positivo. A falta de tais restrições é problemática para aplicações de processamento e troca de dados, que precisam ter código para verificar os tipos de elementos e atributos.
- É difícil usar o mecanismo de DTD para especificar conjuntos de subelementos não ordenados. A ordem raramente é importante para a troca de dados (ao contrário do layout do documento, em que ela é fundamental). Embora a combinação da alternância (o operador |) com a operação * ou +, como na Figura 10.7, permita a especificação de coleções de tags não ordenadas, é muito mais difícil especificar que cada tag só pode aparecer uma vez.
- Existe uma falta de tipos nas `ID`s e `IDREF`s. Assim, não existe um meio de especificar o tipo do elemento ao qual um atributo `IDREF` ou `IDREFS` deve se referir. Como resultado, a DTD da Figura 10.8 não impede que o atributo `proprietários` de um elemento `conta` se refira a outras contas, embora isso não faça sentido.

XML Schema

Um esforço para reparar as deficiências do mecanismo de DTD resultou no desenvolvimento de uma linguagem de esquema mais sofisticada, a **XMLSchema**. Oferecemos uma rápida visão geral da XMLSchema, e depois relacionamos algumas áreas em que ela melhora as DTDs.

XMLSchema define uma série de tipos internos, como `string`, `integer`, `decimal`, `date` e `boolean`. Além disso, ela permite o uso de tipos definidos pelo usuário; estes podem ser tipos simples, com restrições adicionais, ou tipos complexos, criados por meio de construtores como `complexType` e `sequence`.

A Figura 10.10 mostra como a DTD da Figura 10.7 pode ser representada pela XMLSchema; descrevemos a seguir os recursos da XMLSchema ilustrados pela figura.

A primeira coisa a observar é que as definições de esquema em XMLSchema são especificadas na sintaxe XML, usando uma série de tags definidas pela XMLSchema. Para evitar conflitos com as tags definidas pelo usuário, iniciamos a tag XMLSchema com o prefixo do namespace "`xs:`"; esse prefixo está associado ao namespace da XMLSchema pela especificação `xmlns:xs` no elemento raiz:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Observe que qualquer prefixo de namespace poderia ser usado no lugar de `xs`; assim, poderíamos substituir todas as

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="banco" type="TipoBanco" />
<xs:element name="conta">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="número_conta" type="xs:string"/>
            <xs:element name="nome_agência" type="xs:string"/>
            <xs:element name="saldo" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="cliente">
            <xs:element name="número_cliente" type="xs:string"/>
            <xs:element name="rua_cliente" type="xs:string"/>
            <xs:element name="cidade_cliente" type="xs:string"/>
</xs:element>
<xs:element name="depositante">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="nome_cliente" type="xs:string"/>
            <xs:element name="número_conta" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="TipoBanco">>
    <xs:sequence>
        <xs:element ref="conta" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="cliente" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="depositante" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Figura 10.10 Versão em XMLSchema da DTD da Figura 10.7.

ocorrências de "xs:" na definição do esquema por "xsd:" sem alterar o significado da definição do esquema. Todos os tipos definidos pela XMLSchema precisam ser iniciados por esse prefixo de namespace.

O primeiro elemento é o elemento raiz banco, cujo tipo é especificado como sendo TipoBanco, que é declarado mais adiante. O exemplo, então, define os tipos dos elementos conta, cliente e depositante. Observe que cada um desses é especificado por um elemento com tag xs:element, cujo corpo contém a definição do tipo.

O tipo de conta é definido como sendo de um tipo complexo, que é especificado ainda mais para consistir em uma seqüência de elementos número_conta, nome_agência e saldo. Qualquer tipo que tenha atributos ou subelementos aninhados precisa ser especificado como sendo um tipo complexo.

Como alternativa, o tipo de um elemento pode ser especificado como um tipo predefinido pelo atributo type; observe como os tipos da XMLSchema xs:string e xs:decimal são usados para restringir os tipos de elementos de dados como número_conta.

Finalmente, o exemplo define o tipo TipoBanco como possuindo zero ou mais ocorrências de conta, cliente e depositante. Observe o uso de ref para especificar a ocorrência de um elemento definido anteriormente. XMLSchema pode definir o número mínimo e máximo de ocorrências dos subelementos usando minOccurs e maxOccurs. O padrão para as ocorrências mínima e máxima é 1, de modo que estas precisam ser especificadas explicitamente para permitir zero ou mais elementos conta, depositante e cliente.

Os atributos são especificados usando a tag xs:attribute. Por exemplo, poderíamos ter definido número_conta como um atributo acrescentando

```
<xs:attribute name = "número_conta"/>
```

dentro da decodificação do elemento conta. A inclusão do atributo use = "required" à especificação de atributo anterior declara que o atributo precisa ser especificado, enquanto o valor-padrão de use é optional. As especificações de atributo apareceriam diretamente sob a especificação complexType delimitadora, mesmo que os elementos estivessem aninhados dentro de uma especificação de seqüência.

Podemos usar o elemento `xs:complexType` para criar tipos complexos nomeados; a sintaxe é igual à que foi usada para o elemento `xs:complexType` na Figura 10.10, exceto que acrescentamos o atributo `name = nomeTipo` ao elemento `xs:complexType`, em que `nomeTipo` é o nome que queremos dar ao tipo. Depois, podemos usar o tipo nomeado para especificar o tipo de um elemento usando o atributo `type`, assim como usamos `xs:decimal` e `xs:string` em nosso exemplo.

Além de definir tipos, um esquema relacional também permite a especificação de restrições. XMLSchema permite a especificação de chaves e referências de chave, correspondendo à definição de chave primária e chave estrangeira em SQL. Em SQL, uma restrição de chave primária ou restrição exclusiva garante que os valores de atributo não se repitam dentro da relação. No contexto da XML, precisamos especificar um escopo dentro do qual os valores são exclusivos e formam uma chave. O `selector` é uma expressão de caminho que define o escopo para a restrição, e declarações `field` especificam os elementos ou atributos que formam a chave. Para especificar que os números de conta formam uma chave para os elementos `conta` sob o elemento raiz `banco`, incluiríamos a seguinte especificação de restrição à definição do esquema:

```
<xs:key name = "chaveConta">
    <xs:selector xpath = "/banco/conta"/>
    <xs:field    xpath = "número_conta"/>
</xs:key>
```

De modo correspondente, uma restrição de chave estrangeira de `depositante` para `conta` pode ser definida da seguinte maneira:

```
<xs:keyref name = "chaveEdepositanteConta"
refer="chaveConta">
    <xs:selector xpath = "/banco/depositante"/>
    <xs:field    xpath = "número_conta"/>
</xs:key>
```

Observe que o atributo `refer` especifica o nome da declaração de chave que está sendo referenciada, enquanto a especificação `field` identifica os atributos referenciando.

XMLSchema oferece vários benefícios em relação às DTDs, e é muito utilizada hoje. Entre os benefícios que vimos nos exemplos anteriores, destacamos estes:

- Ela permite que o texto que aparece nos elementos seja restrito a tipos específicos, como tipos numéricos em formatos específicos ou tipos complexos, como sequências de elementos de outros tipos.
- Permite a criação de tipos definidos pelo usuário.
- Permite restrições de exclusividade e de chave estrangeira.
- É integrada a namespaces, para permitir que diferentes partes de um documento estejam em conformidade com diferentes esquemas.

Além dos recursos que vimos, a XMLSchema admite vários outros recursos que as DTDs não possuem, como estes:

- Ela permite que os tipos sejam restringidos para criar tipos especializados, por exemplo, especificando valores mínimo e máximo.
- Permite que os tipos complexos sejam estendidos usando uma forma de herança.

Nossa descrição da XMLSchema é apenas uma visão geral; para descobrir mais sobre a XMLSchema, consulte as referências nas notas bibliográficas.

Consulta e transformação

Dada a quantidade cada vez maior de aplicações que utilizam XML para trocar, mediar e armazenar dados, ferramentas para o gerenciamento eficaz de dados XML estão se tornando cada vez mais importantes. Em particular, as ferramentas para consulta e transformação de dados XML são essenciais para extrair informações de grandes corpos de dados XML e converter os dados entre diferentes representações (esquemas) em XML. Assim como a saída de uma consulta relacional é uma relação, a saída de uma consulta XML pode ser um documento XML. Como resultado, consulta e transformação podem ser combinadas em uma única ferramenta.

Várias linguagens oferecem capacidades de consulta e transformação cada vez maiores:

- XPath é uma linguagem para expressões de caminho e, na realidade, é um bloco de montagem para as duas linguagens de consulta restantes.
- XQuery é a linguagem padrão para consultar dados XML. Ela é baseada na SQL, mas é significativamente diferente, pois precisa lidar com dados XML aninhados. XQuery também incorpora expressões XPath.
- XSLT foi elaborada para ser uma linguagem de transformação, como parte do sistema de folha de estilo XSL, usado para controlar a formatação de dados XML para HTML ou outras linguagens de impressão ou exibição. Embora criada para formatação, XSLT pode gerar XML como saída, e pode expressar muitas consultas interessantes. Ela atualmente é a linguagem mais disponível para a manipulação de dados XML, embora XQuery seja mais apropriada para manipulação de banco de dados.

Um **modelo de árvore** dos dados XML é utilizado em todas essas linguagens. Um documento XML é modelado como uma **árvore**, com **nós** correspondendo aos elementos e atributos. Os nós de elemento podem ter nós **filhos**, que podem ser subelementos ou atributos do elemento. Da

mesma forma, cada nó (seja ele atributo ou elemento), que não seja o elemento raiz, possui um nó pai, que é um elemento. A ordem dos elementos e atributos no documento XML é modelada pela ordenação dos filhos dos nós da árvore. Os termos pai, filho, ancestral, descendente e irmãos são interpretados no modelo de árvore dos dados XML.

O conteúdo de texto de um elemento pode ser modelado como um filho de nó de texto do elemento. Elementos contendo texto separado por subelementos intercalados podem ter vários filhos de nó de texto. Por exemplo, um elemento contendo "este é um <bold> maravilhoso </bold> livro" teria um subelemento filho correspondente ao elemento bold e dois filhos de nó de texto correspondentes a "este é um" e "livro". Como tais estruturas normalmente não são usadas em dados de banco de dados, vamos considerar que os elementos não contêm texto e subelementos.

XPath

XPath endereça partes de um documento XML por meio de expressões de caminho. A linguagem pode ser vista como uma extensão das expressões de caminho simples nos bancos de dados orientados a objeto e objeto relacional (ver seção "Identidade de objeto e tipos de referência na SQL" do Capítulo 9). A versão atual do padrão XPath é a XPath 2.0, e nossa descrição é baseada nessa versão.

Uma **expressão de caminho** em XPath é uma sequência de etapas de localização separadas por "/" (em vez do operador "." que separa as etapas no SQL:1999). O resultado de uma expressão de caminho é um conjunto de nós. Por exemplo, no documento da Figura 10.9, a expressão XPath

```
/banco-2/cliente/nome_cliente
```

retornaria estes elementos:

```
<nome_cliente>Joe</nome_cliente>
<nome_cliente>Lisa</nome_cliente>
<nome_cliente>Mary</nome_cliente>
```

A expressão

```
/banco-2/customer/nome_cliente/text( )
```

retornaria os mesmos nomes, mas sem as tags delimitadoras.

As expressões de caminho são avaliadas da esquerda para a direita. Assim como uma hierarquia de diretórios, a '/' inicial indica a raiz do documento. (Observe que essa é uma raiz abstrata "acima" de <banco-2>, que é a tag do documento.)

Enquanto uma expressão de caminho é avaliada, o resultado do caminho em qualquer ponto consiste em um conjunto ordenado de nós a partir do documento. Inicialmente, o conjunto "atual" de elementos contém apenas um nó, a raiz abstrata. Quando a próxima etapa na expressão de caminho é um nome de elemento, como `cliente`, o resultado da etapa consiste nos nós correspondentes aos elementos do nome especificado que sejam filhos dos elementos do conjunto de elementos atual. Esses nós, então, se tornam o conjunto de elementos atual para a próxima etapa da avaliação da expressão de caminho. O resultado de uma expressão de caminho é, então, o conjunto de nós após a última etapa da avaliação da expressão de caminho. Os nós retornados por etapa aparecem na mesma ordem que no documento.

Como vários filhos podem ter o mesmo nome, o número de nós no conjunto de nós pode aumentar ou diminuir a cada etapa. Os valores de atributo também podem ser acessados usando o símbolo "@". Por exemplo, `/banco-2/conta/@número_conta` retorna um conjunto de todos os valores dos atributos `número_conta` dos elementos `conta`. Por padrão, links `IDREF` não são seguidos; veremos como lidar com `IDREFs` mais adiante.

XPath admite diversos outros recursos:

- Predicados de seleção podem vir após qualquer etapa em um caminho, e estão contidos entre colchetes. Por exemplo:

  ```
  /banco-2/conta[saldo > 400]
  ```

  retorna os elementos `conta` com um valor de `saldo` maior que 400, enquanto

  ```
  /banco-2/conta[saldo > 400]/@número_conta
  ```

  retorna os números de conta dessas contas.

  Podemos testar a existência de um subelemento listando-o sem qualquer operação de comparação; por exemplo, se removêssemos apenas "> 400" do exemplo anterior, a expressão retornaria números de conta de todas as contas que possuem um subelemento `saldo`, independente do seu valor.
- XPath oferece várias funções que podem ser usadas como parte de predicados, incluindo o teste da posição do nó atual na ordem de irmão e a função agregada `count( )`, que conta o número de nós combinados pela expressão à qual é aplicada. Por exemplo, a expressão de caminho

  ```
  /banco-2/conta[count(./cliente) > 2]}
  ```

  retorna contas com mais de dois clientes. Os conectivos booleanos `and` e `or` podem ser usados em predicados, enquanto a função `not(...)` pode ser usada para negação.
- A função `id`("algo") retorna o nó (se houver) com um atributo do tipo `ID` e valor "algo". A `id` da função pode ainda ser aplicada a conjuntos de referências, ou mesmo a strings contendo várias referências separadas por espaços, como `IDREFS`. Por exemplo, o caminho

```
/banco-2/conta/id(@proprietário)
```

retorna todos os clientes referenciados a partir do atributo `proprietários` dos elementos `conta`.

- O operador | permite que os resultados da expressão sejam unidos. Por exemplo, se a DTD de `banco-2` também tivesse elementos para empréstimos, com o atributo `mutuário` do tipo `IDREFS` identificando o mutuário do empréstimo, a expressão

```
/banco-2/conta/id(@proprietário) |
/banco-2/empréstimo/id(@mutuário)
```

da aos clientes contas ou empréstimos. Porém, o operador | não pode ser aninhado dentro de outros operadores. Também vale a pena observar que os nós na união são retornados na ordem em que aparecem no documento.

- Uma expressão XPath pode saltar vários níveis de nós usando "//". Por exemplo, a expressão `/banco-2//nome_cliente` encontra todos os elementos `nome_cliente` *em qualquer lugar* sob o elemento `/banco-2`, independentemente dos elementos em que estão contidos e de quantos níveis de elementos delimitadores estão presentes entre os elementos `banco-2` e `nome_cliente`. Esse exemplo ilustra a capacidade de encontrar os dados exigidos sem conhecimento total do esquema.
- Uma etapa no caminho não precisa apenas selecionar pelos filhos dos nós no conjunto de nós atual. Na verdade, essa é apenas uma das várias direções ao longo das quais uma etapa no caminho pode prosseguir, como pais, irmãos, ancestrais e descendentes. Omitimos os detalhes, mas observe que "//", descrito anteriormente, é uma forma abreviada para especificar "todos os descendentes", enquanto ".." especifica o pai.
- A função interna `doc(nome)` retorna a raiz de um documento nomeado; o nome poderia ser um nome de arquivo ou um URL. A raiz retornada pela função pode, então, ser usada em uma expressão de caminho para acessar o conteúdo do documento. Assim, uma expressão de caminho pode ser aplicada a um documento especificado, em vez de ser aplicada ao documento padrão atual.

Por exemplo, se os dados de banco em nosso exemplo de banco estiverem contidos em um arquivo "banco.xml", a expressão de caminho a seguir retornaria todas as contas no banco.

```
doc("banco.xml")/banco/conta
```

A função `collection(name)` é semelhante a `doc`, mas retorna uma coleção de documentos identificados por `name`.

XQuery

O World Wide Web Consortium (W3C) desenvolveu a XQuery como uma linguagem de consulta padrão para XML. Nossa discussão é baseada no rascunho mais recente do padrão da linguagem, disponível no início de janeiro de 2005; embora o padrão final possa diferir, esperamos que os principais recursos abordados aqui permaneçam inalterados. A linguagem XQuery deriva de uma linguagem de consulta da XML, chamada Quilt; a própria Quilt inclui recursos de linguagens anteriores, como XPath, discutida na seção "Xpath", e duas outras linguagens de consulta da XML, XQL e XMQL.

Expressões FLWOR

Consultas XQuery são modeladas a partir de consultas SQL, mas diferem significativamente da SQL. Elas são organizadas em cinco seções: **for**, **let**, **where**, **order by** e **return**. Elas são referenciadas como expressões "FLWOR" (pronuncia-se como em "flower"), com as letras na FLWOR indicando as cinco seções.

Uma expressão FLWOR simples que retorna os números de conta para contas-correntes é baseada no documento XML da Figura 10.9, que usa `ID` e `IDREFS`:

```
for $x in /banco-2/conta
let $numconta := $x/@número_conta
where $x/saldo > 400
return <número_conta> { $numconta } </número_conta>
```

A cláusula **for** é como a cláusula **from** da SQL, e especifica variáveis que variam pelos resultados das expressões XPath. Quando mais de uma variável é especificada, os resultados incluem o produto cartesiano dos valores possíveis que as variáveis podem assumir, assim como na cláusula **from** da SQL.

A cláusula **let** simplesmente permite que os resultados das expressões XPath sejam atribuídos a nomes de variável, para simplificar a representação. A cláusula **where**, assim como a cláusula **where** da SQL, realiza testes adicionais sobre as tuplas unidas a partir da cláusula **for**. A cláusula **order by**, assim como a cláusula **order by** da SQL, permite a classificação da saída. Finalmente, a cláusula **return** permite a construção dos resultados em XML.

Uma consulta FLWOR não precisa conter todas as cláusulas; por exemplo, uma consulta pode conter apenas as cláusulas **for** e **return**, e omitir as cláusulas **let**, **where** e **order by**. A consulta XQuery anterior não possuía uma cláusula **order by**, e a variável `$numconta` na cláusula **return** poderia ser substituída por `$x/@número_conta`. Observe também que, como a cláusula **for** usa expressões XPath, as seleções podem ocorrer dentro da expressão XPath. Assim, uma consulta equivalente pode ter apenas cláusulas **for** e **return**:

```
for $x in /banco-2/conta[saldo > 400]
return <número_conta> { $x/@número_conta }
</número_conta>
```

Porém, a cláusula **let** ajuda a simplificar consultas complexas. Observe também que as variáveis atribuídas por cláusulas **let** podem conter seqüências com vários elementos ou valores, se a expressão de caminho no lado direito retornar uma seqüência de vários elementos ou valores.

Observe o uso de chaves ("{ }") na cláusula **return**. Quando a XQuery encontra um elemento como `<número_conta>` iniciando uma expressão, ela trata seu conteúdo como texto XML normal, exceto pelas partes dentro das chaves, que são avaliadas como expressões. Assim, se omitissemos as chaves na cláusula **return** anterior, o resultado teria várias cópias da string "`$x/@número_conta`", cada uma delimitada em uma tag `número_conta`. O conteúdo dentro das chaves, porém, é tratado como expressões a serem avaliadas. Observe que essa convenção se aplica mesmo que as chaves apareçam dentro das aspas. Assim, poderíamos modificar a consulta anterior para retornar um elemento com a tag `conta`, com o número de conta como um atributo, substituindo a cláusula **return** pelo seguinte:

```
return <conta  número_conta="{$x/@número_conta}" />
```

A XQuery oferece outra maneira de construir elementos usando os construtores **element** e **attribute**. Por exemplo, se a cláusula **return** na consulta anterior fosse substituída pela cláusula **return** a seguir, a consulta retornaria elementos `conta` com `número_conta` e `nome_agência` como atributos e `saldo` como um subelemento.

```
return element conta {
    attribute número_conta {$x/@número_conta},
    attribute nome_agência {$x/nome_agência},
    element saldo {$x/saldo}
}
```

Observe que, como antes, as chaves precisam tratar uma string como uma expressão a ser avaliada.

Joins

Joins (junções) são especificadas em XQuery da mesma forma que na SQL. A junção de elementos `depositante`, `conta` e `cliente` na Figura 10.1, que escrevemos em XSLT na seção "XSLT", pode ser escrita em XQuery da seguinte maneira:

```
for $a in /banco/conta,
    $c in /banco/cliente,
    $d in /banco/depositante
where $a/número_conta = $d/número_conta
    and $c/nome_cliente = $d/nome_cliente
return <conta_cli> { $c $a } </conta_cli>
```

A mesma consulta pode ser expressa com as seleções especificadas como seleções XPath:

```
for $a in /banco/conta,
    $c in /banco/cliente,
    $d in /banco/depositante[número_conta =
$a/número_conta
        and nome_cliente = $c/nome_cliente]
return <conta_cli> { $c $a } </conta_cli>
```

Expressões de caminho em XQuery são iguais às expressões de caminho na XPath 2.0. As expressões de caminho podem retornar um único valor ou elemento, ou uma seqüência de valores ou elementos. Na ausência de informações de esquema, pode não ser possível deduzir se uma expressão de caminho retorna um único valor ou uma seqüência de valores. Tais expressões de caminho podem participar em operações de comparação como =, < e >=.

XQuery possui uma definição interessante das operações de comparação nas seqüências. Por exemplo, a expressão `$x/saldo > 400` tem a interpretação normal se o resultado de `$x/saldo` for um único valor; mas, se o resultado for uma seqüência contendo diversos valores, a expressão é avaliada como verdadeira se pelo menos um dos valores for maior que 400. De modo semelhante, a expressão `$x/saldo = $y/saldo` é avaliada como verdadeira se qualquer um dos valores retornados pela primeira expressão for igual a qualquer um dos valores retornados pela segunda expressão. Se esse comportamento não for apropriado, os operadores `eq`, `ne`, `lt`, `gt`, `le`, `ge` podem ser usados em seu lugar. Estes geram um erro se uma das entradas for uma seqüência com diversos valores.

Consultas aninhadas

Expressões FLWOR da XQuery podem ser aninhadas na cláusula **return**, a fim de gerar aninhamentos de elementos que não aparecem no documento de origem. Esse recurso é semelhante às subconsultas aninhadas, na cláusula **from** das consultas SQL da seção "Aninhando e desaninhando" do Capítulo 9.

Por exemplo, a estrutura XML mostrada na Figura 10.4, com elementos `conta` aninhados dentro de elementos `cliente`, pode ser gerada a partir da estrutura da Figura 10.1 por meio desta consulta:

```
<banco-1> {
    for $c in /banco/cliente
    return
        <cliente>
            { $c/* }
            { for $d in /banco/depositante[nome_cliente
= $c/nome_cliente],
                $a in
/banco/account[número_conta=$d/número_conta]
            return $a }
        </cliente>
} </banco-1>
```

A consulta também introduz a sintaxe `$c/*`, que se refere a todos os filhos do nó (ou seqüência de nós) vinculado à va-

riável $c. De modo semelhante, $c/text( ) fornece o conteúdo de texto de um elemento, sem as tags.

XQuery oferece uma série de funções agregadas, como sum( ) e count( ), que podem ser aplicadas sobre seqüências de elementos ou valores. A função distinct-values( ) aplicada sobre uma seqüência retorna uma seqüência sem duplicação. A seqüência (coleção) de valores retornados por uma expressão de caminho pode ter alguns valores repetidos, pois são repetidos no documento, embora um resultado de expressão XPath possa conter no máximo uma ocorrência de cada nó no documento. XQuery admite muitas outras funções; consulte as referências nas notas bibliográficas para obter mais informações. Na realidade, essas funções são comuns à XPath 2.0 e XQuery, e podem ser usadas em qualquer expressão de caminho XPath.

Para evitar conflitos de namespace, as funções são associadas a um namespace

```
http://www.w3.org/2004/10/xpath-functions
```

que tem um prefixo de namespace padrão, fn. Assim, essas funções podem ser referenciadas sem qualquer ambigüidade como fn:sum ou fn:count.

Embora a XQuery não ofereça uma construção **group by**, consultas agregadas podem ser escritas usando as funções agregadas no caminho ou expressões FLWOR aninhadas dentro da cláusula **return**. Por exemplo, a consulta a seguir no esquema XML banco encontra o saldo total em todas as contas pertencentes a cada cliente.

```
for $c in /banco/cliente
return
   <saldo-total-cliente>>
      <nome_cliente> { $c/nome_cliente }
</nome_cliente>
      <saldo_total> { fn:sum(
         for $d in /banco/depositante[nome_cliente =
$c/nome_cliente],
             $a in /banco/account[número_conta =
$d/número_conta]
         return $a/saldo
      ) } </saldo_total>
   </saldo-total-cliente>
```

Classificação dos resultados

Os resultados podem ser classificados em XQuery por meio da cláusula **order by**. Por exemplo, esta consulta informa todos os elementos cliente classificados pelo subelemento nome_cliente:

```
for $c in /banco/cliente,
order by $c/nome_cliente
return <cliente> { $c/* } </cliente>
```

Para classificar em ordem decrescente, podemos usar **order by** nome_cliente **descending**.

A classificação pode ser feita em vários níveis de aninhamento. Por exemplo, podemos obter uma representação aninhada das informações bancárias classificadas por ordem de nome de cliente, com as contas em cada cliente sendo classificadas por número de conta, da seguinte forma:

```
<banco-1> {
   for $c in /banco/cliente
   order by $c/nome_cliente
   return
      <cliente>
         { $c/* }
         { for $d in /banco/depositante[nome_cliente =
$c/nome_cliente],
                $a in /banco/conta[número_conta =
$d/número_conta]
          order by $a/número_conta
          return <conta> { $a/* } </conta> }
      </cliente>
} </banco-1>
```

Funções e tipos

XQuery oferece diversas funções embutidas, como funções numéricas e funções de combinação e manipulador de strings. Além disso, XQuery admite funções definidas pelo usuário. A seguinte função definida pelo usuário retorna uma lista de todos os saldos de um cliente com um nome especificado:

```
define function saldos(xs:string $c) as xs:decimal* {
   for $d in /banco/depositante[nome_cliente = $c],
       $a in /banco/conta[número_conta =
$d/número_conta]
   return $a/saldo
}
```

As especificações de tipo para argumentos de função e valores de retorno são opcionais, e podem ser omitidas. XQuery usa o sistema de tipo da XMLSchema. O prefixo de namespace xs: utilizado no exemplo anterior é predefinido pela XQuery para ser associado ao namespace XMLSchema.

Os tipos podem ter um sufixo * para indicar uma seqüência de valores desse tipo; por exemplo, a definição da função saldos especifica o valor de retorno como uma seqüência de valores numéricos. Os tipos podem ser especificados parcialmente; por exemplo, o tipo element permite elementos com qualquer tag, enquanto element(conta) permite elementos com a tag conta.

XQUery realiza a conversão de tipo automaticamente sempre que isso for exigido. Por exemplo, se um valor numérico representado por uma string for comparado com um tipo numérico, a conversão de tipo de string para o tipo numérico é feita automaticamente. Quando um elemento é passado a uma função que espera um valor de string, a conversão de tipo para uma string é feita concatenando todos os

valores de texto contidos (aninhados) dentro do elemento. Assim, a função `contains(a,b)`, que verifica se a string `a` contém a string `b`, pode ser usada com seu primeiro argumento definido como um elemento, quando verifica se o elemento `a` contém a string `b` aninhada em qualquer lugar dentro dela. XQuery também oferece funções para converter entre tipos. Por exemplo, `number(x)` converte uma string em um número.

Outros recursos

XQuery oferece uma série de outros recursos, como construções if-then-else, que podem ser usadas dentro de cláusulas **return**, e quantificação existencial e universal, que pode ser usada em predicados nas cláusulas **where**. Por exemplo, a quantificação existencial pode ser expressa na cláusula **where** usando

```
some $e in caminho satisfies P
```

onde `caminho` é uma expressão de caminho e `P` é um predicado que pode usar `$e`. A quantificação universal pode ser expressa usando **every** no lugar de **some**.

Como você pode ver pela descrição anterior, XQuery com XPath é uma linguagem um tanto complexa, e precisa lidar com dados tendo uma estrutura complexa. Embora já tendo se passado muitos anos desde que foi definida inicialmente (como uma especificação de rascunho), muitas implementações ou implementam um subconjunto de XQuery ou são ineficientes sobre grandes conjuntos de dados.

O padrão **XQJ** oferece uma API para submeter consultas XQuery a um sistema de banco de dados XML e recuperar os resultados XML. Sua funcionalidade é semelhante à API JDBC.

XSLT**

Uma **folha de estilo** é uma representação das opções de formatação para um documento, normalmente armazenadas fora do próprio documento, de modo que a formatação é separada do conteúdo. Por exemplo, uma folha de estilo para HTML poderia especificar a fonte a ser usada em todos os cabeçalhos e, assim, substituir uma grande quantidade de declarações de fonte na página HTML. A **XML Stylesheet Language** (XSL) foi projetada originalmente para gerar HTML a partir da XML e, portanto, é uma extensão lógica das folhas de estilo HTML. A linguagem inclui um mecanismo de transformação de uso geral, chamado **XSL Transformations (XSLT)**, que pode ser usado para transformar um documento XML em outro documento XML, e para outros formatos como HTML.[1]

1. O padrão XSL consiste em XSLT e um padrão para especificar recursos de formatação como fontes, margens de página e tabelas. A formatação não é relevante do ponto de vista do banco de dados, de modo que não explicaremos isso aqui.

As transformações XSLT são muito poderosas e, de fato, a XSLT pode até mesmo atuar como uma linguagem de consulta.

As transformações XSLT são expressas como uma série de regras recursivas, chamadas **templates**. Em sua forma básica, as templates permitem a seleção de nós em uma árvore XML por uma expressão XPath. Porém, as templates também podem gerar novo conteúdo XML, de modo que a seleção e a geração de conteúdo podem ser misturadas de maneiras naturais e poderosas. Embora a XSLT possa ser usada como uma linguagem de consulta, sua sintaxe e semântica são muito diferentes da SQL.

Uma template simples para XSLT consiste em uma parte **match** e uma parte **select**. Considere este código XSLT:

```
<xsl:template match="/banco-2/cliente">
    <xsl:value-of select="nome_cliente"/>
</xsl:template>
<xsl:template match="*"/>
```

A instrução `xsl:template` match contém uma expressão XPath que seleciona um ou mais nós. A primeira template combina com elementos `cliente` que ocorrem como filhos do elemento raiz `banco-2`. A instrução `xsl:value-of` delimitada na instrução `match` gera valores a partir dos nós no resultado da expressão XPath. A primeira template gera o valor do subelemento `nome_cliente`; observe que o valor não contém a tag do elemento.

Observe que a segunda template combina com todos os nós. Isso é necessário porque o comportamento-padrão da XSLT sobre elementos do documento de entrada que não combinam com qualquer template é copiar seu atributo e conteúdo de texto para o documento de saída, e aplicar as templates recursivamente aos seus subelementos.

Qualquer texto ou tag na folha de estilo XSLT que não esteja no namespace `xsl` é copiado sem alteração para a saída. A Figura 10.11 mostra como usar esse recurso para fazer com que o nome de cada cliente em nosso exemplo apareça como um subelemento de um elemento "`<cliente>`", colocando a instrução `xsl:value-of` entre `<cliente>` e `</cliente>`. A criação de um atributo, como `id_cliente` no elemento `cliente` gerado, é mais intricada e exige o uso de `xsl:attribute`; veja mais detalhes em um manual de XSLT.

A **recursão estrutural** é uma parte essencial da XSLT. Lembre-se de que os elementos e os subelementos formam naturalmente uma estrutura de árvore. A idéia da recursão estrutural é esta: quando um modelo combinar com um elemento na estrutura de árvore, a XSLT poderá usar a recursão estrutural para aplicar regras de modelo recursivamente sobre subárvores, em vez de simplesmente gerar um valor. Ela aplica as regras recursivamente pela diretiva `xsl:apply-templates`, que aparece dentro de outros modelos.

Por exemplo, os resultados de nossa consulta anterior po-

```
<xsl:template match="banco-2/cliente">
    <cliente>
    <xsl:value-of select="nome_cliente>
    <cliente>
</xsl:template>
<xsl:template match="*"/>
```

Figura 10.11 Usando XSLT para envolver resultados em novos elementos XML.

dem ser colocados em um elemento `<clientes>` que os envolve, acrescentando-se uma regra com `xsl: apply-templates`, como na Figura 10.12. A nova regra combina a tag "banco" exterior e constrói um documento de resultado aplicando todas as outras templates às subárvores que aparecem dentro do elemento `banco`, mas envolvendo os resultados no elemento `<clientes> </clientes>` indicado. Sem a recursão forçada pela cláusula `<xsl:apply-templates/>`, a template geraria `<clientes> </clientes>` e depois aplicaria as outras templates nos subelementos.

De fato, a recursão estrutural é essencial para a construção de documentos XML bem formados, pois os documentos XML precisam ter um único elemento de alto nível contendo todos os outros elementos no documento.

A XSLT oferece um recurso chamado **chaves**, que permite a pesquisa de elementos usando valores de subelementos ou atributos; os objetivos são semelhantes ao da função `id( )` na XPath, mas o recurso de chaves da XSLT permite a utilização de atributos que não sejam atributos `ID`. As chaves são definidas por uma diretiva `xsl:key`, que possui três partes, por exemplo:

```
<xsl:key name="numconta" match="conta"
    use="número_conta"/>
```

O atributo `name` é usado para distinguir as diferentes chaves. O atributo `match` especifica a que nós a chave se aplica. Finalmente, o atributo `use` especifica a expressão a ser utilizada como valor da chave. Observe que a expressão não precisa ser exclusiva a um elemento; ou seja, mais de um elemento poderá ter o mesmo valor de expressão. No exemplo, a chave chamada `numconta` especifica que o subelemento `número_conta` de `conta` deverá ser usado como uma chave para essa conta.

As chaves podem ser usadas subseqüentemente nas templates como parte de qualquer padrão por meio da função `key`. Essa função apanha o nome da chave e um valor, retornando o conjunto dos nós que correspondem a esse valor. Assim, o nó XML para a conta "A-401" pode ser indicado como `key("numconta", "A-401")`.

As chaves podem ser usadas para implementar alguns tipos de junções, como na Figura 10.13. O código na figura pode ser aplicado aos dados XML no formato da Figura 10.1. Aqui, a função `key` faz a junção dos elementos `depo-`

```
<xsl:template match="/banco">
    <clientes>
    <xsl:apply-templates/>
    </clientes>
</xsl:template>
<xsl:template match="/cliente">
    <cliente>
    <<xsl:value-of select="nome_cliente"/>
    </cliente>
</xsl:template>
<xsl:template match="*"/>
```

Figura 10.12 Aplicando regras recursivamente.

```
<xsl:key name="numconta" match="conta" use="número_conta"/>
<xsl:key name="numcli" match="cliente" use="nome_cliente"/>
<xsl:template match="depositante">
    <conta_cli>
    <xsl:value-of select=key("numcli", "nome_cliente")/>
    <xsl:value-of select=key("numconta", "número_conta")/>
    </conta_cli>
</xsl:template>
<<xsl:template match="*"/>
```

Figura 10.13 Joins em XSLT.

sitante com os elementos cliente e conta correspondentes. O resultado da consulta consiste em pares de elementos cliente e conta delimitados por elementos conta_cli.

A XSLT permite que os nós sejam classificados. Um exemplo simples mostra como xsl:sort seria usada em nossa folha de estilo para retornar elementos cliente classificados por nome:

```
<xsl:template match="/banco">
   <xsl:apply-templates select="cliente">
   <xsl:sort select="nome_cliente"/>
   </xsl:apply-templates>
</xsl:template>
<xsl:template match="cliente">
   <cliente>
      <xsl:value-of select="nome_cliente"/>
      <xsl:value-of select="rua_cliente"/>
      <xsl:value-of select="cidade_cliente"/>
   </cliente>
</xsl:template>
<xsl:template match="*"/>
```

Aqui, o elemento xsl:apply-template possui um atributo select, que o restringe a ser aplicado apenas sobre subelementos cliente. A diretiva xsl:sort dentro do elemento xsl:apply-template faz com que os nós sejam classificados *antes* que sejam processados pelo próximo conjunto de templates. Existem opções para permitir a classificação sobre vários subelementos/atributos, por valor numérico, e na ordem decrescente.

Interfaces de programa de aplicação para XML

Com a grande aceitação da XML como um formato para representação e troca de dados, existem muitas ferramentas de software disponíveis para manipulação de dados XML. Existem dois modelos-padrão para a manipulação programática da XML, cada um disponível para uso com diversas linguagens de programação populares. Essas duas APIs podem ser usadas para analisar um documento XML e criar uma representação do documento na memória. Elas são usadas para aplicações que lidam com documentos XML individuais. Porém, observe que elas não são adequadas para consultar grandes coleções de dados XML; mecanismos de consulta declarativos, como XPath e XQuery, são mais adequados para essa tarefa.

Uma das APIs padrão para manipular XML é baseada no *modelo de objeto de documento* (DOM – Document Object Model), que trata o conteúdo XML como uma árvore, com cada elemento representado por um nó, chamado DOMNode. Os programas podem acessar partes do documento de uma forma navegacional, começando com a raiz.

Existem bibliotecas DOM disponíveis para a maioria das linguagens de programação e que estão presentes nos navegadores Web, em que podem ser usadas para manipular o documento exibido ao usuário. Esboçamos aqui algumas das interfaces e métodos na API Java para DOM, para oferecer o estilo DOM.

- A API DOM da Java oferece uma interface chamada Node e interfaces Element e Attribute, que herdam da interface Node.
- A interface Node oferece métodos como getParentNode( ), getFirstChild( ) e getNextSibling( ), para navegar pela árvore DOM começando pelo nó raiz.
- Os subelementos de um elemento podem ser acessados por nome, usando getElementsByTagName(nome), que retorna uma lista de todos os elementos filho com um nome de tag especificado; os membros individuais da lista podem ser acessados pelo método item(i), que retorna o elemento de ordem *i* da lista.
- Os valores de atributo de um elemento podem ser acessados por nome, usando o método getAttribute(nome).
- O valor de texto de um elemento é modelado como um nó Text, que é um filho do nó de elemento. Um nó de elemento sem subelementos possui apenas um nó filho desse tipo. O método getData( ) sobre o nó Text retorna o conteúdo de texto.

O DOM também oferece uma série de funções para atualizar o documento acrescentando e excluindo filhos de atributo e elemento de um nó, definindo valores de nó e assim por diante.

Muito mais detalhes são necessários para escrever um programa DOM real; veja as notas bibliográficas para obter referências a mais informações.

DOM pode ser usado para acessar dados XML armazenados em bancos de dados, e um banco de dados XML pode ser embutido com DOM como sua interface principal para acessar e modificar dados. Porém, a interface DOM não admite qualquer forma de consulta declarativa.

A segunda interface de programação comumente utilizada, a *Simple API for XML* (SAX), é um modelo de *evento*, criado para oferecer uma interface comum entre analisadores e aplicações. Essa API é baseada na noção de *manipuladores de eventos*, que consistem em funções especificadas pelo usuário, associadas a eventos de analise. Os eventos de análise correspondem ao reconhecimento de partes de um documento; por exemplo, um evento é gerado quando a tag de início é encontrada para um elemento, e outro evento é gerado quando a tag de fim é encontrada. As partes de um documento sempre são encontradas na ordem, do início ao fim.

O desenvolvedor de aplicação SAX cria funções de manipulador para cada evento e as registra. Quando um documento é lido pelo analisador SAX, à medida que cada evento ocorre, a função de manipulador é chamada com parâ-

metros descrevendo o evento (como tag de elemento ou conteúdo de texto). As funções do manipulador, então, executam sua tarefa. Por exemplo, para construir uma árvore representando os dados XML, as funções de manipulador para um evento de início de atributo ou elemento poderiam acrescentar um nó (ou nós) a uma árvore construída parcialmente. Os manipuladores de evento da tag de início e fim também teriam de registrar o nó atual na árvore em que novos nós precisam ser anexados; o evento de início do elemento definiria o novo elemento como o nó em que outros nós filhos devem ser anexados. O evento de fim do elemento correspondente definiria o pai do nó como o nó atual onde outros nós filhos devem ser anexados.

SAX geralmente exige mais esforço de programação do que DOM, mas ajuda a evitar o trabalho adicional de criar uma árvore DOM em situações em que a aplicação precisa criar sua própria representação de dados. Se DOM fosse usado para tais aplicações, haveria gastos desnecessários de espaço e tempo para a construção da árvore DOM.

Armazenamento de dados XML

Muitas aplicações exigem o armazenamento de dados XML. Um modo de armazenar dados XML é armazená-los como documentos em um sistema de arquivo, enquanto um segundo é montar um banco de dados de uso especial para armazenar dados XML. Outra técnica é converter os dados XML para uma representação relacional e armazená-los em um banco de dados relacional. Várias alternativas para armazenar dados XML são esboçadas rapidamente nesta seção.

Depósitos de dados não relacionais

Existem várias alternativas para armazenar dados XML em sistemas de armazenamento de dados não relacionais:

- **Armazenar em arquivos simples.** Como a XML é principalmente um formato de arquivo, um mecanismo de armazenamento natural é simplesmente um arquivo simples. Essa técnica tem muitas das desvantagens, esboçadas no Capítulo 1, de usar sistemas de arquivo como base para aplicações de banco de dados. Em particular, ela não possui o isolamento de dados, atomicidade, acesso concorrente e segurança. Porém, a grande disponibilidade de ferramentas de XML que funcionam sobre os dados do arquivo torna relativamente fácil acessar e consultar dados XML armazenados em arquivos. Assim, esse formato de armazenamento pode ser suficiente para algumas aplicações.
- **Criar um banco de dados XML.** Os bancos de dados XML são bancos de dados que utilizam XML como seu modelo de dados básico. Os primeiros bancos de dados XML implementam o Document Object Model em um banco de dados orientado a objeto baseado em C++. Isso permite que grande parte da infra-estrutura do banco de dados orientado a objeto seja reutilizada, enquanto oferece uma interface XML padrão. O acréscimo de XQuery ou outras linguagens de consulta XML oferece consulta declarativa. Outras implementações têm montado toda a infra-estrutura de armazenamento e consulta XML em cima de um gerenciador de armazenamento que oferece suporte transacional.

Embora tenham sido criados vários bancos de dados projetados especificamente para armazenar dados XML, a criação de um sistema de banco de dados cheio de recursos desde o início é uma tarefa muito complexa. Tal banco de dados precisa admitir não apenas o armazenamento e a consulta de dados XML, mas também outros recursos de banco de dados, como transações, segurança, suporte para acesso aos dados pelos clientes e uma série de facilidades de administração. Faz sentido usar, em vez disso, um sistema de banco de dados existente para oferecer essas facilidades, e implementar armazenamento e consulta de dados XML ou em cima da abstração relacional ou como uma camada paralela à abstração relacional. Estudamos essas técnicas na próxima seção.

Bancos de dados relacionais

Como os bancos de dados relacionais são bastante utilizados nas aplicações, existe muito benefício no armazenamento de dados XML em bancos de dados relacionais, de modo que os dados possam ser acessados pelas aplicações.

A conversão de dados XML para o formato relacional normalmente é simples, se os dados foram gerados a partir de um esquema relacional em primeiro lugar e a XML foi utilizada simplesmente como um formato de troca de dados para dados relacionais. Porém, existem muitas aplicações em que os dados XML não são gerados a partir de um esquema relacional, e a tradução dos dados para o formato relacional para armazenamento pode não ser direta. Em particular, os elementos aninhados e os elementos que se repetem (correspondentes a atributos com valor de conjunto) complicam o armazenamento de dados XML no formato relacional. Várias técnicas alternativas estão disponíveis, as quais descrevemos a seguir.

Armazenar como string

Pequenos documentos XML podem ser armazenados como valores de string (**clob**) em tuplas de um banco de dados relacional. Grandes documentos XML com o ele-

mento de nível superior possuindo muitos filhos podem ser tratados pelo armazenamento de cada elemento filho como uma string em uma tupla separada. Por exemplo, os dados XML da Figura 10.1 poderiam ser armazenados como um conjunto de tuplas em um relação *elementos(dados)*, com os *dados* do atributo de cada tupla armazenando um elemento XML (`conta`, `cliente` ou `depositante`) no formato de string.

Embora essa representação seja fácil de usar, o sistema de banco de dados não conhece o esquema dos elementos armazenados. Como resultado, não é possível consultar os dados diretamente. Na verdade, nem sequer é possível implementar seleções simples, como encontrar todos os elementos `conta`, ou encontrar o elemento `conta` com número de conta A-401, sem percorrer todas as tuplas da relação e examinar o conteúdo de string.

Uma solução parcial para esse problema é armazenar diferentes tipos de elementos em diferentes relações, e também armazenar os valores de alguns elementos críticos como atributos da relação para ativar a indexação. Por exemplo, em nosso exemplo, as relações seriam *elementos_conta*, *elementos_cliente* e *elementos_depositante*, cada um com um *dado* de atributo. Cada relação pode ter atributos extras para armazenar os valores de alguns subelementos, como *número_conta* ou *nome_cliente*. Assim, uma consulta que exige elementos `conta` com um número de conta especificado pode ser respondida de forma eficiente com essa representação. Essa técnica depende da informação de tipo sobre dados XML, como a DTD dos dados.

Alguns sistemas de banco de dados, como Oracle, admitem **índices de função**, que podem ajudar a evitar a replicação de atributos entre a string XML e os atributos da relação. Ao contrário dos índices normais, que são sobre valores de atributo, os índices de função podem ser baseados no resultado da aplicação de funções definidas pelo usuário sobre as tuplas. Por exemplo, um índice de função pode ser baseado em uma função definida pelo usuário que retorna o valor do subelemento `número_conta` da string XML em uma tupla. O índice pode, então, ser usado da mesma maneira que um índice sobre um atributo *número_conta*.

As técnicas anteriores têm a desvantagem de que uma grande parte da informação XML é armazenada dentro de strings. É possível armazenar todas as informações das relações de uma das várias maneiras que examinamos a seguir.

Representação de árvore

Dados XML quaisquer podem ser modelados como uma árvore e armazenados usando um par de relações:

nós(id, tipo, rótulo, valor)
filho(id-filho, id-pai)

Cada elemento e atributo nos dados XML recebe um identificador exclusivo. Uma tupla inserida na relação *nós* para cada elemento e atributo com seu identificador (*id*), seu tipo (atributo ou elemento), o nome do elemento ou atributo (*rótulo*) e o valor de texto do elemento ou atributo (*valor*). A relação *filho* é usada para registrar o elemento pai de cada elemento e atributo. Se a informação de ordem dos elementos e atributos tiver de ser preservada, uma *posição* de atributo extra pode ser acrescentada à relação *filho* para indicar a posição relativa do filho entre os filhos do pai. Como exercício, você pode representar os dados XML da Figura 10.1 usando essa técnica.

Essa representação tem a vantagem de que todas as informações XML podem ser representadas diretamente em formato relacional, e muitas consultas XML podem ser traduzidas para consultas relacionais e executadas dentro do sistema de banco de dados. Porém, ela tem a desvantagem de que cada elemento é desmembrado em muitas partes, e muitas associações são necessárias para transformar os subelementos de volta para um elemento.

Mapa de relações

Nessa técnica, os elementos XML cujo esquema é conhecido são mapeados para relações e atributos. Os elementos cujo esquema é desconhecido são armazenados como strings ou como uma árvore.

Uma relação é criada para cada tipo de elemento (incluindo subelementos) cujo esquema é conhecido e cujo tipo é um tipo complexo (ou seja, contém atributos ou subelementos). Os atributos da relação são definidos da seguinte maneira:

- Todos os atributos desses elementos são armazenados como atributos com valor de string da relação.
- Se um subelemento do elemento for um tipo simples (ou seja, não puder ter atributos ou subelementos), um atributo é acrescentado à relação para representar o subelemento. O tipo do atributo da relação é, como padrão, um valor de string, mas, se o subelemento tiver um tipo XMLSchema, um tipo SQL correspondente poderá ser usado.

  Por exemplo, o subelemento *número_conta* do elemento *conta* torna-se um atributo da relação *conta*.
- Caso contrário, uma relação é criada correspondendo ao subelemento (usando as mesmas regras recursivamente em seus subelementos). Além do mais,
  - Um atributo identificador é acrescentado às relações representando o elemento. (O atributo identificador é acrescentado apenas uma vez, mesmo que um elemento tenha vários subelementos.)
  - Um atributo *id_pai* é acrescentado à relação representando o subelemento, armazenando o identificador de seu elemento pai.

- Se a ordenação tiver de ser preservada, um atributo *posição* é acrescentado à relação representando o subelemento.

Observe que, quando aplicamos essa técnica aos elementos abaixo do elemento raiz da DTD dos dados da Figura 10.1, recebemos de volta o esquema relacional original que usamos nos capítulos anteriores.

Outras variantes dessa técnica também são possíveis. Por exemplo, as relações correspondentes aos subelementos que podem ocorrer no máximo uma vez podem ser "achatadas" na relação pai pela mudança de todos os atributos para a relação pai. As notas bibliográficas oferecem referências a diferentes técnicas para representar dados XML como relações.

Publicando e fragmentando dados XML

Quando a XML é usada para trocar dados entre aplicações corporativas, os dados normalmente são originados nos bancos de dados relacionais. Os dados nos bancos de dados relacionais precisam ser *publicados*, ou seja, convertidos para o formato XML, para exportar para outras aplicações. Os dados que chegam precisam ser *fragmentados*, ou seja, convertidos de volta de XML para o formato de relação normalizada e armazenados em um banco de dados relacional. Embora o código de aplicação possa realizar as operações de publicação e fragmentação, as operações são tão comuns que as conversões devem ser feitas automaticamente, sem escrever o código da aplicação, onde possível. Os vendedores de banco de dados realizaram muito esforço para fazer com que seus produtos de banco de dados trabalhem com XML.

Um banco de dados preparado para XML aceita um mecanismo automático para publicar dados relacionais como XML. O mapeamento utilizado para publicar dados pode ser simples ou complexo. Uma relação simples com o mapeamento XML poderia criar um elemento XML para cada linha de uma tabela e tornar cada coluna dessa linha um subelemento do elemento XML. O esquema XML da Figura 10.1 pode ser criado a partir de uma representação relacional de informações de banco, usando tal mapeamento. Esse tipo de mapeamento é simples de se gerar automaticamente. Essa visão SQL dos dados relacionais pode ser tratada como um documento XML *virtual*, e as consultas XML podem ser executadas contra o documento XML virtual.

Um mapeamento mais complicado permitiria a criação de estruturas aninhadas. As extensões da SQL com consultas aninhadas na cláusula **select** foram desenvolvidas para permitir a criação fácil da saída XML aninhada. Essas extensões são esboçadas na seção "SQL/XML".

Os mapeamentos também precisam ser definidos para fragmentar dados XML em uma representação relacional. Para os dados XML criados a partir de uma representação relacional, o mapeamento exigido para compartilhar os dados é um inverso direto do mapeamento utilizado para publicar os dados. Para o caso geral, um mapeamento pode ser gerado conforme esboçado na seção anterior.

Armazenamento nativo dentro de um banco de dados relacional

Mais recentemente, os bancos de dados relacionais começaram a admitir o **armazenamento nativo** da XML. Tais sistemas armazenam os dados XML como strings ou em representações binárias mais eficientes, sem converter os dados para o formato relacional. Um novo tipo de dado **xml** é introduzido para representar dados XML, embora os tipos de dados CLOB e BLOB possam oferecer o mecanismo de armazenamento básico. As linguagens de consulta XML, como XPath e XQuery, são utilizadas para consultar dados XML.

Uma relação com um atributo do tipo **xml** pode ser usada para armazenar uma coleção de documentos XML; cada documento é armazenado como um valor do tipo **xml** em uma tupla separada. Os índices de uso especial são criados para indexar os dados XML.

Diversos sistemas de banco de dados oferecem suporte nativo para dados XML. Eles oferecem um tipo de dado **xml** e permitem que consultas XQuery sejam embutidas dentro de consultas SQL. Uma consulta XQuery pode ser executada sobre um único documento XML e pode ser embutida dentro de uma consulta SQL, para permitir que seja executada em cada um de uma coleção de documentos, com cada documento armazenado em uma tupla separada. Por exemplo, veja na seção "Suporte a XML no SQL Server 2005" no Capítulo 29 mais detalhes sobre o suporte nativo no Microsoft SQL Server 2005.

SQL/XML

O padrão SQL/XML, definido recentemente, estabelece uma extensão padrão da SQL, permitindo a criação de saída XML aninhada. O padrão possui várias partes, incluindo um modo padrão de mapear tipos SQL para tipos XMLSchema, e um modo padrão de mapear esquemas relacionais para esquemas XML, além de extensões da linguagem de consulta SQL.

Por exemplo, a representação SQL/XML da relação *conta* teria um esquema XML com o elemento mais externo *conta*, com cada tupla mapeada para um elemento XML *linha*, e cada atributo de relação mapeado para um elemento XML do mesmo nome (com algumas convenções para resolver incompatibilidades com caracteres especiais nos nomes).

```
<banco>
<conta>
   <linha>
        <número_conta> A-101 </número_conta>
        <nome_agência> Downtown </nome_agência>
        <saldo> 500 </saldo>
   </linha>
   <linha>
        <número_conta> A-102 </número_conta>
        <nome_agência> Perryridge </nome_agência>
        <saldo> 400 </saldo>
   </linha>
   <linha>
        <número_conta> A-201 </número_conta>
        <nome_agência> Brighton </nome_agência>
        <saldo> 900 </saldo>
   </linha>
   <</conta>
  <cliente>
   <linha>
        <nome_cliente> Johnson </nome_cliente>
        <rua_cliente> Alma </rua_cliente>
        <cidade_cliente> Palo Alto </cidade_cliente>
  </linha>
   <linha>
        <nome_cliente> Hayes </nome_cliente>
        <rua_cliente> Main </rua_cliente>
        <cidade_cliente> Harrison </cidade_cliente>
       </linha>
     </cliente>
  <depositante>
        <linha>
        <número_conta> A-101 </número_conta>
        <nome_cliente> Johnson </nome_cliente>
   </linha>
   <linha>
        <número_conta> A-201 </número_conta>
        <nome_cliente> Johnson </nome_cliente>
   </linha>
   <linha>
        <número_conta> A-102 </número_conta>
        <nome_cliente>> Hayes </nome_cliente>
        </linha>
  </depositante>
</banco>
```

Figura 10.14 Representação SQL/XML das informações bancárias.

Um esquema SQL inteiro, com várias relações, também pode ser mapeado para XML de uma forma semelhante. A Figura 10.14 mostra a representação SQL/XML do esquema *banco* contendo as relações *conta*, *cliente* e *depositante*.

SQL/XML acrescenta vários operadores e agrega operações à SQL para permitir a construção da saída XML diretamente a partir da SQL estendida. A função **xmlelement** pode ser usada para criar elementos XML, enquanto **xmlattributes** pode ser usado para criar atributos, conforme ilustrado pela consulta a seguir.

```
select xmlelement( name "conta",
        xmlattributes( número-conta as número_conta),
        xmlelement( name "nome_agência", nome-agência),
        xmlelement( name "saldo", saldo))
    from conta
```

Essa consulta cria um elemento XML para cada conta, com o número de conta representado como um atributo, e o nome da agência e o saldo como subelementos. O resultado se pareceria com os elementos de conta mostrados na Figura 10.9, sem o atributo proprietários. O operador

xmlattributes cria o nome de atributo XML usando o nome de atributo SQL, que pode ser alterado usando uma cláusula **as**, conforme mostramos.

O operador **xmlforest** simplifica a construção de estruturas XML. Sua sintaxe e comportamento são semelhantes àqueles de **xmlattributes**, exceto por criar uma floresta (coleção) de subelementos, em vez de uma lista de atributos. São necessários vários argumentos, criando um elemento para cada argumento, com o nome SQL do atributo utilizado como um nome de elemento XML. O operador **xmlconcat** pode ser usado para concatenar elementos criados por subexpressões em uma floresta.

Quando o valor SQL utilizado para construir um atributo é nulo, o atributo é omitido. Os valores nulos são omitidos quando o corpo de um elemento é construído.

A SQL/XML também oferece uma nova função agregada **xmlagg**, que cria uma floresta (coleção) de elementos XML a partir da coleção de valores sobre a qual é aplicada. A consulta a seguir cria um elemento para cada agência, contendo como subelementos todos os números de conta nessa agência. Como a consulta tem uma cláusula **group by** *nome-agência*, a função agregada é aplicada a todas as contas nessa agência, criando uma seqüência de elementos de número de conta.

```
select xmlelement( name "agência",
nome_agência,
xmlagg ( xmlforest(número_conta)
order by número_conta))
from conta
group by nome-agência
```

A SQL/XML permite que a seqüência criada por **xmlagg** seja classificada, conforme ilustramos na consulta anterior. Veja, nas notas bibliográficas, referências a outras informações sobre SQL/XML.

Aplicações XML

Agora, esboçamos várias aplicações da XML para armazenar e comunicar (trocar) dados e para acessar Web services (recursos de informação).

Armazenando dados com estrutura complexa

Muitas aplicações precisam armazenar dados que são estruturados, mas que não são modelados com facilidade como relações. Considere, por exemplo, as preferências do usuário que precisam ser armazenadas por uma aplicação, como um navegador. Normalmente, existem muitos campos (por exemplo, página principal, configurações de segurança, configurações de linguagem e configurações de exibição) que precisam ser registrados. Alguns dos campos possuem múltiplos valores, por exemplo, uma lista de sites confiáveis, ou talvez listas ordenadas, como uma lista de marcadores. As aplicações tradicionalmente usavam algum tipo de representação textual para armazenar tais dados. Hoje, muitas dessas aplicações preferem armazenar essa informação de configuração em formato XML. As representações textuais ocasionais usadas anteriormente exigem esforço para projetar e criar analisadores que possam ler o arquivo e converter os dados para um formato que um programa possa utilizar. A representação XML evita essas duas etapas.

Representações baseadas em XML até mesmo foram propostas como padrões para armazenar documentos, dados de planilha e outros dados que fazem parte dos pacotes de aplicação para escritório.

XML também é usada para representar dados com estrutura complexa, que precisam ser trocados entre diferentes partes de uma aplicação. Por exemplo, um sistema de banco de dados pode representar um plano de execução de consulta (uma expressão de álgebra relacional com informações extras sobre como executar operações) utilizando XML. Isso permite que uma parte do sistema gere o plano de execução da consulta e outra parte o apresente, sem usar uma estrutura de dados compartilhada. Por exemplo, os dados podem ser gerados em um sistema servidor e enviados para um sistema cliente onde os dados serão exibidos.

Formatos padronizados para troca de dados

Padrões baseados em XML para representação de dados têm sido desenvolvidos para diversas aplicações especializadas, desde aplicações de negócios, como sistemas bancários e de entrega, até aplicações científicas, como química e biologia molecular. Alguns exemplos são:

- A indústria química precisa de informações sobre produtos químicos, como sua estrutura molecular, e uma série de propriedades importantes, como pontos de ebulição e fusão, valores calorífico e solubilidade em vários solventes. *ChemML* é um padrão para representar tais informações.
- No setor de entrega, transportadoras de bens e oficiais de alfândega e impostos precisam de registros de remessa contendo informações detalhadas sobre os bens sendo remetidos, desde quem e onde eles foram enviados, até para quem e onde eles serão entregues, o valor monetário dos produtos, e assim por diante.
- Um mercado on-line, em que as empresas podem comprar e vender produtos (um mercado chamado de business-to-business – B2B), exige informações como catálogos de produtos, incluindo descrições detalhadas de

produtos e informações de preço, estoques de produtos, cotas para uma venda proposta e ordens de compra. Por exemplo, os padrões *RosettaNet* para aplicações de e-business definem esquemas e semântica XML para representar dados e padrões para troca de mensagens.

O uso de esquemas relacionais normalizados para modelar tais requisitos de dados complexos resultaria em uma grande quantidade de relações que não correspondem diretamente aos objetos que estão sendo modelados. As relações normalmente teriam grandes quantidades de atributos; a representação explícita de nomes de atributo/elemento junto com os valores em XML ajuda a evitar confusão entre atributos. As representações de elemento aninhadas ajudam a reduzir a quantidade de relações que precisam ser representadas, além do número de junções exigidas para obter as informações solicitadas, ao custo possível da redundância. Por exemplo, em nosso exemplo bancário, listar clientes com elementos `conta` aninhados dentro de elementos `conta`, como na Figura 10.4, resulta em um formato que é mais natural para algumas aplicações – em particular, para humanos lerem – do que a representação normalizada da Figura 10.1.

Web services

As aplicações normalmente exigem dados de fora da organização, ou de outro departamento na mesma organização, que utiliza um banco de dados diferente. Em muitas dessas situações, a organização ou departamento externo não deseja permitir acesso direto ao seu banco de dados usando SQL, mas deseja oferecer formas limitadas de informação por meio de interfaces predefinidas.

Quando a informação tiver de ser usada diretamente por um humano, as organizações oferecem formatos baseados na Web, em que os usuários podem inserir valores e retornar informações desejadas em formato HTML. Porém, existem muitas aplicações em que tais informações precisam ser acessadas por programas de software, em vez de usuários finais. Fornecer os resultados de uma consulta em formato XML é um requisito claro. Além disso, faz sentido especificar os valores de entrada para a consulta também em formato XML.

Com efeito, o provedor das informações define procedimentos cuja entrada e saída estão em formato XML. O protocolo HTTP é utilizado para comunicar as informações de entrada e saída, pois é muito utilizado e pode passar por firewalls que as instituições utilizam para afastar o tráfego indesejado da Internet.

O **Simple Object Access Protocol (SOAP)** define um padrão para invocar procedimentos, usando XML para representar a entrada e a saída do procedimento. SOAP define um esquema XML padrão para representar o nome do procedimento, e indicadores de status do resultado, como indicadores de falha/erro. Os parâmetros e resultados de procedimento são dados XML dependentes da aplicação, embutidos dentro dos cabeçalhos XML do SOAP.

Normalmente, HTTP é usado como o protocolo de transporte para SOAP, mas um protocolo baseado em mensagem (como o e-mail baseado no protocolo SMTP) também pode ser utilizado. O padrão SOAP é bastante comum hoje. Por exemplo, Amazon e Google oferecem procedimentos baseados em SOAP para executar atividades de busca e outras. Esses procedimentos podem ser invocados por outras aplicações que oferecem serviços de nível mais alto aos usuários. O padrão SOAP é independente da linguagem de programação básica, e é possível que um site trabalhando com uma linguagem, como C#, invoque um serviço executado em uma linguagem diferente, como Java.

Um site oferecendo tal coleção de procedimentos SOAP é chamado de **Web service**. Diversos padrões foram definidos para dar suporte aos Web services. A **Web Services Description Language (WSDL)** é uma linguagem utilizada para descrever as capacidades de um Web service. WSDL oferece facilidades que as definições de interface (ou definições de função) oferecem em uma linguagem de programação tradicional, especificando quais funções estão disponíveis e seus tipos de entrada e saída. Além disso, WSDL permite a especificação do URL e número de porta de rede a serem usados para invocar o Web service. Há também um padrão chamado **Universal Description, Discovery, and Integration (UDDI)**, que define como um catálogo de Web services disponíveis pode ser criado e como um programa pode pesquisar no diretório para encontrar um Web service que satisfaça seus requisitos.

O exemplo a seguir ilustra o valor dos Web services. Uma companhia aérea pode definir um Web service oferecendo um conjunto de procedimentos que podem ser invocados por um site de viagem; estes podem incluir procedimentos para localizar horários de vôo e informações de preço, além de fazer reservas de vôo. O site de viagem pode interagir com vários Web services, fornecidos por diferentes companhias aéreas, hotéis e outras empresas, para oferecer informações de viagem a um cliente e fazer reversas de viagem. Com o suporte dos Web services, as empresas individuais permitem que um serviço útil seja construído com base neles, integrando os serviços individuais. Os usuários podem interagir com um único site para fazer suas reservas de viagem, sem ter de entrar em contato com vários sites separados.

Para invocar um Web service, um cliente precisa preparar uma mensagem XML SOAP apropriada e enviá-la ao serviço; ao receber o resultado codificado em XML, o cliente precisa então extrair informações do resultado em XML.

Existem APIs padrão em linguagens como Java e C# para criar e extrair informações das mensagens SOAP.

Consulte as notas bibliográficas para obter referências a mais informações sobre Web services.

Mediação de dados

A comparação de compras é um exemplo de uma aplicação de mediação, em que os dados sobre itens, estoque, preços e custos de entrega são extraídos de uma série de sites oferecendo um item em particular para venda. A informação agregada resultante é muito mais valiosa do que a informação individual oferecida por um único site.

Um gerenciador financeiro pessoal é uma aplicação semelhante no contexto bancário. Considere um consumidor com diversas contas para gerenciar, como contas-correntes, contas de aplicações e contas de aposentadoria. Suponha que essas contas possam ser mantidas em diferentes instituições. Oferecer gerenciamento centralizado para todas as contas de um cliente é um desafio importante. A mediação baseada em XML resolve o problema extraindo uma representação XML de informações de conta dos respectivos sites das instituições financeiras em que o indivíduo mantém contas. Essa informação pode ser facilmente extraída, se a instituição a exportar em um formato XML padrão, por exemplo, como um Web service. Para as que não fazem isso, um software de *wrapper* é usado para gerar dados XML a partir das páginas Web em HTML retornadas pelo site. Aplicações wrapper precisam de manutenção constante, pois elas dependem dos detalhes de formatação das páginas Web, que mudam com freqüência. Apesar disso, o valor fornecido pela mediação normalmente justifica o esforço exigido para desenvolver e manter wrappers.

Quando as ferramentas básicas estão disponíveis para extrair informações de cada origem, uma aplicação *mediadora* é usada para combinar as informações extraídas sob um único esquema. Isso pode exigir mais transformação dos dados XML de cada site, pois diferentes sites podem estruturar a mesma informação de formas diferentes. Por exemplo, um dos bancos pode exportar informações no formato da Figura 10.1, enquanto outro pode usar o formato aninhado da Figura 10.4. Eles também podem usar diferentes nomes para a mesma informação (por exemplo, `número_conta` e `id_conta`), ou podem até mesmo usar o mesmo nome para informações diferentes. O mediador precisa decidir a respeito de um esquema que represente toda a informação exigida, e precisa oferecer código para transformar os dados entre as diferentes representações. Essas questões são discutidas com mais detalhes na seção "Bancos de dados distribuídos heterogêneos" no Capítulo 22, no contexto dos bancos de dados distribuídos. As linguagens de consulta XML, como XSLT e XQuery, desempenham um papel importante na tarefa de transformação entre diferentes representações da XML.

Resumo

- Assim como a Hyper-Text Markup Language (HTML), em que a Web é baseada, a Extensible Markup Language (XML) é descendente da Standard Generalized Markup Language (SGML). XML foi intencionada originalmente para oferecer marcação funcional para documentos Web, mas agora se tornou o formato de dados padrão para a troca de dados entre aplicações.
- Documentos XML contêm elementos com tags inicial e final combinando, indicando o início e o fim de um elemento. Os elementos podem ter subelementos aninhados dentro deles, até qualquer nível de aninhamento. Os elementos também podem ter atributos. A escolha entre representar informações como atributos e subelementos normalmente é arbitrária no contexto da representação de dados.
- Os elementos podem ter um atributo do tipo `ID`, que armazena um identificador exclusivo para o elemento. Os elementos também podem armazenar referências a outros elementos, usando atributos do tipo `IDREF`. Os atributos do tipo `IDREFS` podem armazenar uma lista de referências.
- Os documentos podem opcionalmente ter seu esquema especificado por uma Document Type Declaration (DTD). A DTD de um documento especifica quais elementos podem ocorrer, como eles podem ser aninhados e que atributos cada elemento pode ter. Embora as DTDs sejam muito utilizadas, elas possuem várias limitações. Por exemplo, elas não oferecem um sistema de tipo.
- XMLSchema agora é o mecanismo padrão para especificar o esquema de um documento XML. Ele oferece um grande conjunto de tipos básicos, além de construções para criar tipos complexos e especificar restrições de integridade, incluindo restrições de chave e restrições de chave estrangeira (keyref).
- Dados XML podem ser representados como estruturas de árvore, com os nós correspondendo aos elementos e atributos. O aninhamento de elementos é refletido pela estrutura pai-filho da representação de árvore.
- Expressões de caminho podem ser usadas para atravessar a estrutura de árvore XML, para localizar os dados solicitados. XPath é uma linguagem padrão para expressões de caminho, e permite que os elementos solicitados sejam especificados por um caminho tipo sistema de arquivos, além de permitir seleções e outros recursos. XPath também forma parte das outras linguagens de consulta XML.
- A XQuery é a linguagem padrão para consultar dados

XML. Ela possui uma estrutura não muito diferente da SQL, com cláusulas **for**, **let**, **where**, **order by** e **return**. Porém, ela admite muitas extensões para lidar com a natureza de árvore da XML e permitir a transformação de documentos XML em outros documentos com uma estrutura significativamente diferente. Expressões de caminho XPath formam uma parte da XQuery. XQuery admite consultas aninhadas e funções definidas pelo usuário.

- A linguagem XSLT foi criada originalmente como a linguagem de transformação para uma facilidade de folha de estilo, em outras palavras, para aplicar informações de formatação aos documentos XML. Entretanto, XSLT oferece recursos muito poderosos para consulta e transformação, e possui grande disponibilidade, de modo que é usada para a consulta de dados XML.
- As APIs DOM e SAX são muito usadas para acesso programático a dados XML. Essas APIs estão disponíveis em diversas linguagens de programação.
- Os dados XML podem ser armazenados de várias maneiras diferentes. Os dados XML também podem ser armazenados em sistemas de arquivo, ou em bancos de dados XML, que utilizam XML como sua representação interna.

  Dados XML podem ser armazenados como strings em um banco de dados relacional. Como alternativa, as relações podem representar dados XML como árvore. Como alternativa, dados XML podem ser mapeados para relações da mesma maneira como esquemas E-R são mapeados para esquemas relacionais. O armazenamento nativo de XML nos bancos de dados relacionais é facilitado pelo acréscimo de um tipo de dado **xml** à SQL.
- XML é usada em diversas aplicações, como armazenamento de dados complexos, troca de dados entre organizações em um formato padronizado, mediação de dados e Web services. Web services oferecem uma interface de chamada de procedimento remoto, com XML sendo o mecanismo para codificar parâmetros e também resultados.

Termos de revisão

- Extensible Markup Language (XML)
- Hyper-Text Markup Language (HTML)
- Standard Generalized Markup Language
- linguagem de marcação
- Tags
- Autodocumentação
- Elemento
- Elemento raiz
- Elementos aninhados
- Atributo
- Namespace
- Namespace padrão
- Definição de esquema
- Document Type Definition (DTD)
  - ☐ `ID`
  - ☐ `IDREF` e `IDREFS`
- XMLSchema
  - ☐ Tipos simples e complexos
  - ☐ Tipo sequence
  - ☐ Key e keyref
  - ☐ Restrições de ocorrência
- Modelo de árvore de dados XML
- Nós
- Consulta e transformação
- Expressões de caminho
- XPath
- XQuery
  - ☐ Expressões FLWOR
    - **for**
    - **let**
    - **where**
    - **order by**
    - **return**
  - ☐ Joins
  - ☐ Expressão FLWOR aninhada
  - ☐ Classificação
- XML Stylesheet Language (XSL)
- XSL Transformations (XSLT)
  - ☐ Templates
    - Match
    - Select
  - ☐ Recursão estrutural
  - ☐ Chaves
  - ☐ Classificação
- API XML
- Document Object Model (DOM)
- Simple API for XML (SAX)
- Armazenamento de dados XML
  - ☐ Em depósitos de dados não relacionais
  - ☐ Em bancos de dados relacionais
    - Armazenamento como string
    - Representação de árvore
    - Mapa de relações
    - Publicar e fragmentar
    - Bancos de dados preparado para XML
    - Armazenamento nativo
    - SQL/XML
- Aplicações XML
  - ☐ Armazenamento de dados complexos
  - ☐ Troca de dados
  - ☐ Mediação de dados
  - ☐ SOAP
  - ☐ Web services

Exercícios práticos

10.1 Dê uma representação alternativa das informações bancárias contendo os mesmos dados da Figura 10.1, mas usando atributos no lugar de subelementos. Dê também a DTD para essa representação.

10.2 Dê a DTD para uma representação XML do seguinte esquema relacional aninhado

Emp = (*enome*, *CjFilhos* setof(*Filhos*), *CjHabilidades* setof(*Habilidades*))
Filhos = (*nome*, *Nascimento*)
Nascimento = (*dia*, *mês*, *ano*)
Habilidades = (*tipo*, *CjExames* setof(*Exames*))
Exames = (*ano*, *cidade*)

10.3 Escreva uma consulta em XPath na DTD do Exercício prático 10.2 para listar todos os tipos de habilidade em *Emp*.

10.4 Escreva uma consulta em XQuery na representação XML da Figura 10.1 para encontrar o saldo total, por todas as contas, em cada agência.

10.5 Escreva uma consulta em XQuery na representação XML da Figura 10.1 para calcular a junção externa esquerda dos elementos `depositante` com elementos `conta`. (Dica: Use a quantificação universal.)

10.6 Escreva consultas em XQuery e XSLT para gerar elementos de cliente com elementos de conta associados, aninhados dentro dos elementos cliente, dada a representação de informação bancária usando `ID` e `IDREFS` da Figura 10.9.

10.7 Dê um esquema relacional para representar informações bibliográficas especificadas conforme o fragmento de DTD da Figura 10.15. O esquema relacional precisa registrar a ordem dos elementos `autor`. Você pode considerar que somente livros e artigos aparecem como elementos de nível superior nos documentos XML.

10.8 Mostre a representação de árvore dos dados XML da Figura 10.1 e a representação da árvore usando as relações de *nós* e *filho* descritas na seção "Bancos de dados relacionais".

10.9 Considere a seguinte DTD recursiva.

```
<!DOCTYPE peças [
    <!ELEMENT peça (nome, infosubpeça*)>
    <!ELEMENT infosubpeça (peça, quantidade)>
    <!ELEMENT nome ( #PCDATA )>
    <!ELEMENT quantidade ( #PCDATA )>
] >
```

a. Dê um pequeno exemplo de dados correspondentes a essa DTD.

b. Mostre como mapear essa DTD para um esquema relacional. Você pode considerar que os nomes de peça são exclusivos; ou seja, sempre que uma peça aparece, sua estrutura de subpeça será a mesma.

c. Crie um esquema em XMLSchema correspondente a essa DTD.

Exercícios

10.10 Mostre, por meio de uma DTD, como representar a relação *livros* não-1NF da seção "Tipos de dados complexos" do Capítulo 9, usando XML.

10.11 Escreva as consultas a seguir em XQuery, considerando a DTD do Exercícioprático 10.2.

a. Encontre os nomes de todos os empregados que possuem um filho que tenha data de nascimento em março.

b. Encontre os empregados que fizeram exame para o tipo de habilidade "digitação" na cidade "Dayton".

c. Liste todos os tipos de habilidade em *Emp*.

10.12 Considere os dados XML mostrados na Figura 10.2. Suponha que queremos encontrar ordens de compra que pediram duas ou mais cópias da peça com identificador 123. Considere a seguinte tentativa de solucionar esse problema:

```
for $p in ordemcompra
where $p/peça/id = 123 and $p/peça/quantidade >= 2
return $p
```

Explique por que a consulta pode retornar algumas ordens de compra que pedem menos de duas cópias da peça 123. Dê uma versão correta dessa consulta.

```
<!DOCTYPE bibliografia [
    <!ELEMENT  livro (título, autor+, ano, editora, local?)>
    <!ELEMENT  artigo (título, autor+, periódico, ano, número, volume, pags?)>
    <!ELEMENT  autor ( último_nome, primeiro_nome) >
    <!ELEMENT título ( #PCDATA )>
    ... declarações PCDATA semelhantes para ano, editora, local, periódico,
        ano, número, volume, pags, último_nome e primeiro_nome
]>
```

Figura 10.15 DTD para dados bibliográficos.

10.13 Dê uma consulta em XQuery para inverter o aninhamento de dados do Exercício 10.10. Ou seja, no nível de aninhamento mais externo, a saída precisa ter elementos correspondentes a autores, e cada um desses elementos precisa ter, dentro dele, itens correspondentes a todos os livros escritos pelo autor.

10.14 Dê a DTD para uma representação XML da informação na Figura 6.31. Crie um tipo de elemento separado para representar cada relacionamento, mas use `ID` e `IDREF` para implementar chaves primária e estrangeira.

10.15 Dê uma representação XMLSchema da DTD do Exercício 10.14.

10.16 Escreva consultas em XQuery sobre o fragmento de DTD de bibliografia da Figura 10.15, para fazer o seguinte.

a. Encontre todos os autores que escreveram um livro e um artigo no mesmo ano.
b. Relacione os livros e os artigos classificados por ano.
c. Relacione os livros com mais de um autor.
d. Encontre todos os livros que contêm a palavra "database" em seu título e a palavra "Hank" no nome (primeiro ou último) de um autor.

10.17 Dê um mapeamento relacional do esquema de ordem de compra XML ilustrado na Figura 10.2, usando a técnica descrita na seção "Mapa de relações".
Sugira como remover a redundância no esquema relacional, se os identificadores de item determinarem funcionalmente a descrição e os nomes de comprador e fornecedor determinarem funcionalmente o endereço de comprador e fornecedor, respectivamente.

10.18 Escreva consultas em SQL/XML para converter dados bancários do esquema relacional que usamos nos capítulos anteriores para os esquemas XML *banco-1* e *banco-2*. (Para o esquema *banco-2*, você pode considerar que a relação do cliente tem um atributo extra *id_cliente*.)

10.19 Como no Exercício 10.18, escreva consultas para converter os dados bancários para os esquemas XML *banco-1* e *banco-2*, mas desta vez escrevendo consultas XQuery no banco de dados SQL/XML padrão para mapeamento XML.

10.20 Um modo de fragmentar um documento XML é usar XQuery para converter o esquema em um mapeamento SQL/XML do esquema relacional correspondente, e depois usar o mapeamento SQL/XML na direção contrária para preencher a relação.
Como ilustração, crie uma consulta XQuery para converter dados do esquema XML *banco-1* para o esquema SQL/XML mostrado na Figura 10.14.

10.21 Considere o esquema XML de exemplo da seção "XML Schema", e escreva consultas XQuery para executar as tarefas a seguir.

a. Verificar se a restrição de chave mostrada na seção "XML Schema" é mantida.
b. Verificar se a restrição keyref mostrada na seção "XML Schema" é mantida.

10.22 Considere o Exercício prático 10.7, e suponha que os autores também pudessem aparecer como elementos de nível superior. Que mudança teria de ser feita no esquema relacional?

Notas bibliográficas

O World Wide Web Consortium (W3C) atua como a agência de padrões para padrões relacionados à Web, incluindo a XML básica e as linguagens relacionadas à XML, como XPath, XSLT e XQuery. Diversos relatórios técnicos, definindo os padrões relacionados à XML, estão disponíveis no endereço `www.w3c.org`. Esse site também possui tutoriais e indicadores de software que implementam os diversos padrões. O site XML Cover Pages (`www.oasis-open.org/cover/`) também possui muitas informações obre XML, incluindo a especificação da linguagem Relax NG, para especificar esquemas XML.

Katz *et al.* [2004] oferecem uma cobertura de livro-texto detalhada sobre XQuery. Quilt é descrita em Chamberlin *et al.* [2000]. Deutsch *et al.* [1999] descrevem a linguagem XML-QL. A integração da consulta por palavra-chave à XML é explicada por Florescu *et al.* [2000] e Amer-Yaha *et al.* [2004].

Funderburk *et al.* [2002a], Florescu e Kossmann [1999] descrevem o armazenamento de dados XML. Eisenberg e Melton [2004a] oferecem uma visão geral da SQL/XML, enquanto Funderburk e outros oferecem visões gerais da SQL/XM e da XQuery.

Schning [2001] descreve um banco de dados projetado para XML. Veja nos Capítulos de 27 a 29 mais informações sobre o suporte da XML nos bancos de dados comerciais. Eisenberg e Melton [2004b] oferecem uma visão geral da API XQJ para XQuery.

Ferramentas

Diversas ferramentas para lidar com XML estão disponíveis em domínio público. O site `www.oasis-open.org/cover/` contém links para diversas ferramentas de software para XML e XSL (incluindo XSLT). O site do W3C (`www.w3c.org`) possui páginas descrevendo os diversos pa-

drões relacionados à XML, além de indicadores para ferramentas de software como implementações de linguagem. Você precisa saber que várias implementações, como Galax, são provas de conceito. Embora podendo servir como ferramentas de aprendizado, elas não são capazes de lidar com bancos de dados grandes. Vários bancos de dados comerciais, incluindo IBM DB2, Oracle e Microsoft SQL Server, aceitam o armazenamento XML, publicação usando várias extensões da SQL e consulta usando XPath e XQuery.

PARTE 4

Armazenamento e consulta de dados

Embora um sistema de banco de dados ofereça uma visão de alto nível dos dados, por fim, os dados precisam ser armazenados como bits em um ou mais dispositivos de armazenamento. Uma grande maioria dos bancos de dados hoje armazena dados em disco magnético e apanha dados para a memória principal, para processamento, ou copia dados para fitas e outros dispositivos de backup, para armazenamento de arquivamento. As características físicas dos dispositivos de armazenamento desempenham um papel importante no modo como os dados são armazenados, principalmente porque o acesso a um dado aleatório no disco é muito mais lento do que o acesso à memória: o acesso ao disco leva dezenas de milissegundos, enquanto o acesso à memória leva um décimo de microssegundo.

O Capítulo 11 começa com uma visão geral do meio de armazenamento físico, incluindo mecanismos para reduzir as chances de perda de dados devido a falhas de dispositivo. O capítulo descreve em seguida como os registros são mapeados em arquivos, que, por sua vez, são mapeados em bits no disco.

Muitas consultas referenciam apenas uma pequena proporção dos registros em um arquivo. Um índice é uma estrutura que ajuda a localizar rapidamente registros desejados de uma relação, sem examinar todos os registros. O índice deste livro-texto é um exemplo, embora, ao contrário dos índices de banco de dados, ele sirva para uso humano. O Capítulo 12 descreve vários tipos de índices utilizados nos sistemas de banco de dados.

As consultas do usuário precisam ser executadas sobre o conteúdo do banco de dados, que reside em dispositivos de armazenamento. Normalmente é conveniente desmembrar as consultas em operações menores, mais ou menos correspondendo às operações da álgebra relacional. O Capítulo 13 descreve como as consultas são processadas, apresentando algoritmos para implementar operações individuais, e depois esboçando como as operações são executadas em sincronia, para processar uma consulta.

Existem muitas maneiras alternativas de processar uma consulta, que podem ter custos bastante variados. A otimização da consulta refere-se ao processo de localizar o método de menor custo para avaliar determinada consulta. O Capítulo 14 descreve o processo de otimização da consulta.

CAPÍTULO 11

Armazenamento e estrutura de arquivos

Nos capítulos anteriores, enfatizamos os modelos de nível mais alto de um banco de dados. Por exemplo, no nível *conceitual* ou *lógico*, visualizamos o banco de dados, no modelo relacional, como uma coleção de tabelas. Na verdade, o modelo lógico do banco de dados é o nível correto para usuários de banco de dados focalizarem. Isso porque o objetivo de um sistema de banco de dados é simplificar e facilitar o acesso aos dados. Os usuários do sistema não devem ser sobrecarregados desnecessariamente com os detalhes físicos da implementação do sistema.

Neste capítulo, porém, e também nos Capítulos 12, 13 e 14, analisamos abaixo dos níveis mais altos, enquanto descrevemos diversos métodos para implementar os modelos de dados e linguagens apresentados nos capítulos anteriores. Começamos com as características dos meios de armazenamento básicos, como sistemas de disco e fita. Depois, definimos diversas estruturas de dados que permitem o acesso rápido aos dados. Consideramos várias estruturas alternativas, cada qual mais adequada a um tipo diferente de acesso aos dados. A opção final da estrutura de dados precisa ser feita com base no uso esperado do sistema e das características físicas da máquina específica.

Visão geral do meio de armazenamento físico

Existem vários tipos de armazenamento de dados na maioria dos sistemas de computador. Esses meios de armazenamento são classificados pela velocidade com que os dados podem ser acessados, pelo custo por unidade de dados para comprar o meio e pela confiabilidade do meio. Entre os meios normalmente disponíveis, destacamos estes:

- **Cache.** O cache é a forma de armazenamento mais rápida e dispendiosa. A memória cache é pequena; seu uso é gerenciado pelo hardware do sistema de computador. Não nos preocuparemos com o gerenciamento do armazenamento em cache no sistema de banco de dados.
- **Memória principal.** O meio de armazenamento utilizado para os dados que estão disponíveis para serem operados é a memória principal. As instruções de máquina de uso geral operam sobre a memória principal. Embora a memória principal possa conter muitos megabytes de dados (um PC típico vem com pelo menos 512 megabytes), ou ainda centenas de gigabytes de dados em grandes sistemas de servidor, ela geralmente é muito pequena (ou muito cara) para armazenar o banco de dados inteiro. O conteúdo da memória principal normalmente se perde se houver falta de energia ou uma falha no sistema.
- **Memória flash.** A memória flash difere da memória principal porque os dados sobrevivem à falta de energia. A leitura de dados da memória flash leva menos de 100 nanossegundos (um nanossegundo é 1/1000 de um microssegundo), o que é aproximadamente a mesma velocidade da leitura de dados da memória principal. Porém, a escrita de dados na memória flash é mais complicada – os dados podem ser escritos uma vez, o que leva cerca de 4 a 10 microssegundos, mas não podem ser reescritos diretamente. Para escrever sobre a memória que já foi escrita, temos de apagar o banco de memória inteiro ao mesmo tempo; depois, ele estará pronto para ser escrito novamente. Uma desvantagem da memória flash é que ela só pode aceitar um número limitado de ciclos de apagamento, variando de 10.000 a 1 milhão. A memória flash é uma forma de *memória somente leitura programável e apagável eletricamente* (*EEPROM*); outras formas

de EEPROM permitem que locais de memória individuais sejam apagados e reescritos, mas esses tipos não são muito utilizados.

A memória flash encontrou popularidade como um substituto para os discos magnéticos, para armazenar pequenos volumes de dados (em 2005, normalmente menos de 1 gigabyte, embora a memória flash de maior capacidade, capaz de armazenar muitos gigabytes, esteja começando a aparecer) em sistemas de computador de baixo custo, como sistemas de computador que estão embutidos em outros dispositivos, em computadores portáteis e em outros dispositivos eletrônicos digitais, como câmeras digitais. (No início de 2005, 256 megabytes de memória flash para uma câmera custam cerca de US$25, enquanto 1 gigabyte custa menos de US$100). A memória flash também é usada em "pendrives USB", que podem ser conectados nos slots **Universal Serial Bus (USB)** dos dispositivos de computação. Esses pendrives USB se tornaram um meio popular de transportar dados entre sistemas de computador ("disquetes" desempenharam o mesmo papel nos tempos antigos, mas sua capacidade limitada os tornou obsoletos).

- **Armazenamento de disco magnético.** O principal meio para armazenamento de dados on-line a longo prazo é o disco magnético. Normalmente, o banco de dados inteiro é armazenado em disco magnético. O sistema precisa mover os dados do disco para a memória principal, de modo que possam ser acessados. Depois que o sistema tiver realizado as operações designadas, os dados que foram modificados precisam ser gravados em disco.

  O tamanho dos discos magnéticos atualmente varia de alguns gigabytes até 400 gigabytes. Um disco de 250 gigabytes custa cerca de US$160 em 2005. A extremidade inferior e superior dessa faixa cresce em cerca de 50% ao ano, e podemos esperar discos de capacidade muito maior a cada ano. O armazenamento de disco sobrevive a faltas de energia e falhas do sistema. Os próprios dispositivos de armazenamento em disco às vezes podem falhar e, portanto, destruir dados, mas essas falhas normalmente ocorrem com muito menos freqüência do que as falhas do sistema.
- **Armazenamento óptico.** As formas mais populares de armazenamento óptico são o *compact disk* (CD), que pode manter cerca de 700 megabytes de dados e possui um tempo de execução de cerca de 80 minutos, e o *digital video disk* (DVD), que pode manter 4,7 ou 8,5 gigabytes de dados por lado do disco (ou até 17 gigabytes em um disco com dois lados). A expansão **digital versatile disk** também é usada no lugar do **digital video disk**, pois os DVDs podem manter qualquer dado digital, e não apenas vídeo. Os dados são armazenados de forma óptica em um disco e são lidos por um laser.

  Os discos ópticos usados nos compact disks de leitura (CDROM) ou digital video disk de leitura (DVDROM) não podem ser gravados, mas são fornecidos com dados pré-gravados. Há também versões para "gravar uma vez" dos compact disks (chamados CD-R) e digital video disk (chamados DVD-R e DVD+R), que podem ser gravados apenas uma vez; esses discos também são chamados de discos **write-once, read-many** (WORM). Há também versões para "múltiplas gravações" em compact disk (chamados CD-RW) e em digital video disk (DVD-RW, DVD+RW e DVD-RAM), que podem ser gravados várias vezes.

  Sistemas de **jukebox** de disco óptico contêm algumas unidades e vários discos que podem ser carregados em uma das unidades automaticamente (por um braço robô) por demanda.
- **Armazenamento em fita.** O armazenamento em fita é usado principalmente para backup e arquivamento de dados. Embora a fita magnética seja mais barata do que os discos, o acesso aos dados é muito mais lento, pois a fita precisa ser acessada seqüencialmente desde o início. Por esse motivo, o armazenamento em fita é chamado de armazenamento por **acesso seqüencial**. Ao contrário, o armazenamento em disco é chamado de armazenamento por **acesso direto**, pois é possível ler dados de qualquer local no disco.

  As fitas possuem uma alta capacidade (atualmente, existem fitas de 40 a 300 gigabytes), e podem ser removidas da unidade de fita, de modo que são bastante adequadas para o armazenamento com arquivamento barato. Bibliotecas de fita (jukeboxes) são usadas para manter coleções de dados excepcionalmente grandes, como dados de satélites, que poderiam incluir até centenas de terabytes (1 terabyte = 10^{12}bytes), ou ainda vários petabytes (1 petabyte = 10^{15} bytes) de dados em alguns casos.

Os diversos meios de armazenamento podem ser organizados em uma hierarquia (Figura 11.1), de acordo com sua velocidade e seu custo. Os níveis mais altos são caros, mas são rápidos. Ao descermos na hierarquia, o custo por bit diminui, enquanto o tempo de acesso aumenta. Essa compensação é razoável; se determinado sistema de armazenamento fosse mais rápido e mais barato do que outro – outras propriedades sendo iguais –, então não haveria motivo para usar a memória mais lenta, mais dispendiosa. Na verdade, muitos dispositivos de armazenamento mais antigos, incluindo fita de papel e memórias de tambor, são relegados aos museus, agora que a fita magnética e a memória de semicondutor se tornaram mais rápidos e mais baratos. As próprias fitas magnéticas eram usadas para armazenar dados ativos quando os discos eram muito caros e tinham pouca capacidade de armazenamento. Hoje, quase todos os

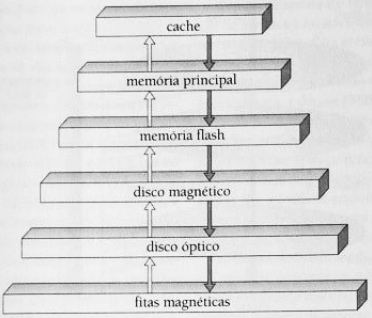


Figura 11.1 Hierarquia do dispositivo de armazenamento.

dados ativos estão armazenados em discos, exceto em casos muito raros, em que são armazenados em fita ou em jukeboxes ópticos.

Os meios de armazenamento mais rápidos – por exemplo, cache e memória principal – são considerados **armazenamento primário**. Os meios no próximo nível da hierarquia – por exemplo, discos magnéticos – são considerados **armazenamento secundário**, ou **armazenamento on-line**. Os meios no nível mais baixo da hierarquia – por exemplo, fita magnética e jukeboxes de discos ópticos – são considerados **armazenamento terciário**, ou **armazenamento offline**.

Além da velocidade e do custo dos diversos sistemas de armazenamento, há também a questão do armazenamento volátil. O **armazenamento volátil** perde seu conteúdo quando a energia ao dispositivo é removida. Na hierarquia mostrada na Figura 11.1, os sistemas de armazenamento da memória principal para cima são voláteis, enquanto os sistemas de armazenamento abaixo da memória principal são não-voláteis. Os dados precisam ser gravados no **armazenamento não volátil** por segurança. Retornaremos a esse assunto no Capítulo 17.

Discos magnéticos

Discos magnéticos oferecem a maior parte do armazenamento secundário para os sistemas de computador modernos. Embora as capacidades de disco tenham crescido ano após ano, os requisitos de armazenamento das grandes aplicações também crescem muito rapidamente, em alguns casos, ainda mais rápido do que a taxa de crescimento das capacidades de disco. Um banco de dados grande pode exigir centenas de discos.

Características físicas dos discos

Fisicamente, os discos são relativamente simples (Figura 11.2). Cada **placa** de disco possui uma forma circular plana. Suas duas superfícies são cobertas com um material magnético, e as informações são gravadas nas superfícies. As placas são feitas de metal rígido ou vidro.

Quando o disco está em uso, o motor da unidade gira em uma velocidade alta constante (normalmente, 60, 90 ou 120 rotações por segundo, mas existem discos rodando a 250 rotações por segundo). Existe uma cabeça de leitura-escrita posicionada logo acima da superfície da placa. A superfície do disco é dividida logicamente em **trilhas**, que são subdivididas em **setores**. Um **setor** é a menor unidade de informação que pode ser lida ou escrita no disco. Nos discos atualmente disponíveis, os tamanhos de setor normalmente são de 512 bytes; existem cerca de 50.000 a 100.000 trilhas por placa, e 1 a 5 placas por disco. As trilhas internas (próximas do eixo) são de menor tamanho, e em discos da geração atual, as trilhas externas contêm mais setores do que as trilhas internas; os números típicos estão em torno de 500 setores por trilha nas trilhas internas, e em torno de 1.000 setores por trilha nas trilhas externas. Os números variam entre diferentes modelos; os modelos de maior capacidade normalmente possuem mais setores por trilha e mais trilhas em cada placa.

A **cabeça de leitura-escrita** armazena informações sobre um setor magneticamente como inversos da direção da magnetização do material magnético.

Cada lado de uma placa de um disco possui uma cabeça de leitura-escrita que se move pela placa para acessar diferentes trilhas. Um disco normalmente contém muitas placas, e as cabeças de leitura-escrita de todas as trilhas são

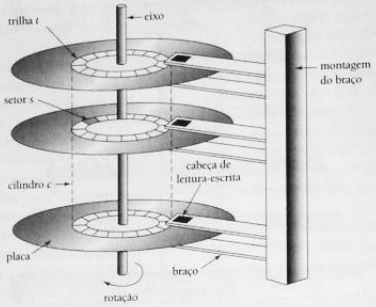


Figura 11.2 Movendo o mecanismo da cabeça do disco.

montadas em um único dispositivo, chamado **braço do disco**, e se movem juntas. As placas de disco montadas em um eixo e as cabeças montadas em um braço do disco são conhecidas juntamente como **montagens cabeça-disco**. Como as cabeças em todas as placas se movem juntas, quando a cabeça em uma placa está na *i*-ésima trilha, as cabeças em todas as outras placas também estão na *i*-ésima trilha de suas respectivas placas. Logo, as *i*-ésimas trilhas de todas as placas juntas são chamadas de *i*-ésimo **cilindro**.

Hoje, os discos com um diâmetro de placa de 3,5 polegadas dominam o mercado. Eles possuem um custo mais baixo e tempos de busca mais rápidos (devido às menores distâncias de busca) do que os discos de diâmetro maior (até 14 polegadas) que eram comuns anteriormente, embora ofereçam capacidade de armazenamento alta. Discos com diâmetros ainda menores são usados em dispositivos portáteis, como computadores de laptop, e alguns computadores de mão e players de música.

As cabeças de leitura-escrita são mantidas o mais próximo possível da superfície do disco, para aumentar a densidade de gravação. A cabeça normalmente flutua ou voa apenas a alguns microns da superfície do disco; o giro do disco cria uma pequena brisa, e a montagem da cabeça é modelada de modo que a brisa mantenha a cabeça flutuando logo acima da superfície do disco. Como a cabeça flutua tão próxima da superfície, as placas precisam ser fabricadas cuidadosamente para que sejam totalmente planas.

Quedas de cabeça podem ser um problema. Se a cabeça entrar em contato com a superfície do disco, a cabeça pode arranhar o meio de gravação no disco, destruindo os dados que estavam lá. Em discos de geração antiga, a cabeça tocando a superfície fazia com que o meio removido voasse e parasse entre as outras cabeças e suas placas, causando mais quedas; assim, uma queda de cabeça poderia resultar em uma falha do disco inteiro. As unidades de disco da geração atual utilizam um fino filme de metal magnético como meio de gravação. Elas são muito menos suscetíveis a falhas pelas quedas de cabeça do que os discos mais antigos, cobertos de óxido.

Uma **controladora de disco** realiza a interface entre o sistema de computador e o hardware real da unidade de disco; em sistemas de disco modernos, a controladora de disco é implementada dentro da unidade de disco. Uma controladora de disco aceita comandos de alto nível para ler ou escrever um setor, e inicia ações, como mover o braço do disco para a trilha correta e realmente ler ou escrever os dados. As controladoras de disco também colocam **somas de verificação** em cada setor gravado; a soma de verificação (check sum) é calculada a partir dos dados gravados no setor. Quando o setor é lido de volta, a controladora calcula a soma de verificação novamente a partir dos dados apanhados e a compara com a soma de verificação armazenada. Se os dados forem adulterados, com uma alta probabilidade, a soma de verificação calculada não combinará com a soma de verificação armazenada. Se houver tal erro, a controladora tentará novamente e lerá várias vezes; se o erro continuar a ocorrer, a controladora sinalizará uma falha de leitura.

Outra tarefa interessante que as controladoras de disco realizam é **remapear os setores defeituosos**. Se a controladora detectar que um setor está danificado quando o disco for formatado inicialmente, ou quando for feita uma tentativa de escrever o setor, ela pode mapear logicamente o setor para um local físico diferente (alocado a partir de um pool de setores extras, reservados para essa finalidade). O

remapeamento é observado no disco ou na memória não-volátil, e a escrita é executada no novo local.

Os discos estão conectados a um sistema de computador por meio de uma interconexão de alta velocidade. Existem várias interfaces comuns para conectar discos a computadores pessoais e estações de trabalho: (1) a interface AT attachment (ATA) (que é uma versão mais rápida da interface Integrated Drive Electronics (IDE) usada anteriormente nos IBM PCs); (2) a nova versão AT, que é SATA (serial ATA; a versão ATA original agora se chama PATA, ou Parallel ATA, para distingui-la da SATA); e (3) a interface Small-Computer-System Interconnect (SCSI; pronuncia-se "scuzzy"). Os sistemas de mainframe e servidor normalmente possuem uma interface mais rápida e mais dispendiosa, como as versões de alta capacidade da interface SCSI, e a interface Fiber Channel. Os sistemas de disco externos portáteis normalmente utilizam a interface USB ou a interface FireWire.

Embora os discos em geral sejam conectados diretamente por cabos à interface de disco do sistema de computador, eles podem estar situados remotamente e ser conectados por uma rede de alta velocidade à controladora de disco. Na arquitetura Storage Area Network (SAN), grandes quantidades de discos são conectadas por uma rede de alta velocidade a uma série de computadores servidores. Os discos normalmente são organizados no local usando uma técnica de organização de armazenamento chamada Redundant Arrays of Independent Disks (RAID) (descrita mais adiante, na seção "RAID"), para dar aos servidores uma visão lógica de um disco muito grande e muito confiável. O computador e o subsistema de disco continuam a usar interfaces SCSI ou Fiber Channel para falar entre si, embora possam estar separados por uma rede. O acesso remoto aos discos por uma SAN significa que os discos podem ser compartilhados por vários computadores que poderiam executar diferentes partes de uma aplicação em paralelo. O acesso remoto também significa que os discos contendo dados importantes podem ser mantidos em uma sala de servidores central, onde podem ser monitorados e mantidos por administradores do sistema, em vez de ficarem espalhados em diferentes partes de uma organização.

Network Attached Storage (NAS) é uma alternativa à SAN. NAS é muito semelhante à SAN, exceto que, em vez do armazenamento em rede aparecer como um grande disco, ele oferece uma interface de sistema de arquivo usando protocolos de sistema de arquivo em rede, como NFS ou CIFS.

Medidas de desempenho dos discos

As principais medidas das qualidades de um disco são capacidade, tempo de acesso, taxa de transferência de dados e confiabilidade.

Tempo de acesso é o tempo desde quando uma solicitação de leitura ou escrita é emitida até quando inicia a transferência de dados. Para acessar (ou seja, ler ou escrever) dados em determinado setor de um disco, o braço primeiro precisa se mover de modo que fique posicionado sobre a trilha correta, e depois precisa esperar até que o setor apareça sob ele enquanto o disco gira. O tempo para reposicionar o braço é chamado de **tempo de busca**, e aumenta com a distância que o braço precisa mover. Os tempos de busca típicos variam de 2 a 30 milissegundos, dependendo da distância que a trilha se encontra da posição inicial do braço. Discos menores costumam ter menores tempos de busca, pois a cabeça precisa trafegar por uma distância menor.

O **tempo de busca médio** é a média dos tempos de busca, medido por uma seqüência de solicitações aleatórias (uniformemente distribuídas). Se todas as trilhas tiverem o mesmo número de setores, e desconsiderarmos o tempo exigido para a cabeça começar a mover e parar de mover, podemos mostrar que o tempo de busca médio é um terço do tempo de busca do pior caso. Levando esses fatores em consideração, o tempo de busca médio fica em torno de metade do maior tempo de busca . Os tempos de busca médios atualmente variam entre 4 e 10 milissegundos, dependendo do modelo do disco.

Quando a cabeça tiver atingido a trilha desejada, o tempo gasto esperando até que o setor a ser acessado apareça sob a cabeça é chamado de **tempo de latência rotacional**. As velocidades rotacionais dos discos de hoje variam de 5400 rotações por minuto (90 rotações por segundo) até 15.000 rotações por minuto (250 rotações por segundo) ou, de modo equivalente, 4 milissegundos a 11,1 milissegundos por rotação. Na média, metade de uma rotação do disco é necessária para que o início do setor desejado apareça sob a cabeça. Assim, o **tempo de latência médio** do disco é metade do tempo para a rotação completa do disco.

O tempo de acesso é, então, a soma do tempo de busca e da latência, e varia de 8 a 20 milissegundos. Quando o primeiro setor dos dados a serem acessados tiver vindo sob a cabeça, a transferência de dados é iniciada. A **taxa de transferência de dados** é a velocidade em que os dados podem ser apanhados ou armazenados no disco. Os sistemas de disco atuais admitem taxas de transferência máximas de 25 a 100 megabytes por segundo; para as trilhas mais internas do disco, as taxas de transferência são muito mais baixas que as taxas de transferência máximas, pois elas possuem menos setores. Por exemplo, um disco com uma taxa de transferência máxima de 100 megabytes por segundo pode ter uma taxa de transferência sustentada de algo em torno de 30 megabytes por segundo em suas trilhas mais internas.

A última medida comumente utilizada de um disco é o **tempo médio para a falha (MTTF – Mean Time To Failure)**, que é uma medida da confiabilidade do disco. O tempo

médio para a falha de um disco (ou de qualquer outro sistema) é a quantidade de tempo que, na média, podemos esperar que o sistema funcione continuamente, sem qualquer falha. De acordo com as afirmações dos fabricantes, o tempo médio para a falha dos discos atualmente varia de 500.000 a 1.200.000 horas – cerca de 57 a 136 anos. Na prática, o tempo médio para a falha é calculado sobre a probabilidade de falha quando o disco é novo – o valor significa que, dados 1.000 discos relativamente novos, se o MTTF é de 1.200.000 horas, na média um deles falhará em 1.200 horas. Um tempo médio para falha de 1.200.000 horas não significa que se pode esperar que o disco funcione por 136 anos! A maioria dos discos tem um tempo de vida esperado de cerca de 5 anos, e taxas de falha significativamente mais altas depois que tiverem mais do que alguns poucos anos.

As unidades de disco para máquinas de desktop normalmente admitem a interface Parallel ATA, que admite taxas de transferência de 133 megabytes por segundo, e a interface Serial ATA (SATA), que admite 150 megabytes por segundo. As interfaces mais antigas ATA-4 e ATA-5 admitiam taxas de transferência de 33 e 66 megabytes por segundo, respectivamente. As unidades de disco projetadas para sistemas servidores normalmente admitem a interface Ultra320 SCSI, que oferece taxas de transferência de até 256 megabytes por segundo. A taxa de transferência de uma interface é compartilhada entre todos os discos conectados à interface (a interface SATA só permite que um disco seja conectado a cada interface).

Otimização de acesso ao bloco de disco

As solicitações de E/S de disco são geradas pelo sistema de arquivos e pelo gerenciador de memória virtual, encontrado na maioria dos sistemas operacionais. Cada solicitação especifica o endereço no disco a ser referenciado; esse endereço está na forma de um *número de bloco*. Um **bloco** é uma unidade lógica consistindo em um número fixo de setores contíguos. Os tamanhos de bloco variam de 512 bytes a vários kilobytes. Os dados são transferidos entre disco e memória principal em unidades de blocos.

O acesso aos dados no disco é de várias ordens de grandeza mais lento do que o acesso aos dados na memória principal. Como resultado, diversas técnicas foram desenvolvidas para melhorar a velocidade de acesso aos blocos no disco. Uma técnica, a colocação de blocos em buffers de memória, para satisfazer solicitações futuras, é discutida na seção "Acesso ao armazenamento". Aqui, discutimos várias outras técnicas.

- **Escalonamento.** Se vários blocos de um cilindro precisam ser transferidos do disco para a memória principal, podemos economizar tempo de acesso solicitando os blocos na ordem em que passarão sob as cabeças. Se os blocos desejados estiverem em cilindros diferentes, é vantajoso solicitar os blocos em uma ordem que reduza o movimento de braço do disco. Os algoritmos de **escalonamento de braço do disco** tentam ordenar os acessos às trilhas em um padrão que aumenta o número de acessos que podem ser processados. Um algoritmo normalmente utilizado é o **algoritmo de elevador**, que funciona da mesma maneira que muitos elevadores. Suponha que, inicialmente, o braço esteja movendo da trilha mais interna para o exterior do disco. Sob o controle dos algoritmos de elevador, para cada trilha para a qual existe uma solicitação de acesso, o braço para nessa trilha, atende as solicitações para a trilha e depois continua movendo para fora, até que não haja mais solicitações aguardando por outras trilhas. Nesse ponto, o braço muda de direção, e se move para o interior, novamente parando em cada trilha para a qual existe uma solicitação, até que alcance a trilha onde não existem mais solicitações de trilhas em direção ao centro. Depois, ele reverte a direção e inicia um novo ciclo. As controladoras de disco normalmente realizam a tarefa de reordenar as solicitações de leitura para melhorar o desempenho, pois estão intimamente cientes da organização dos blocos no disco, da posição rotacional das placas de disco e da posição do braço do disco.
- **Organização de arquivo.** Para reduzir o tempo de acesso ao bloco, podemos organizar os blocos no disco de uma maneira que corresponde de perto ao modo como esperamos que os dados sejam acessados. Por exemplo, se esperarmos que um arquivo seja acessado seqüencialmente, então o ideal é manter todos os blocos do arquivo seqüencialmente em cilindros adjacentes. Os sistemas operacionais mais antigos, como os sistemas operacionais de mainframes IBM, davam aos programadores o controle minucioso do posicionamento dos arquivos, permitindo que um programador reserve um conjunto de cilindros para armazenar um arquivo. Porém, esse controle coloca um peso sobre o programador ou administrador do sistema para decidir, por exemplo, quantos cilindros alocar para um arquivo, e pode exigir uma reorganização dispendiosa se os dados forem inseridos ou excluídos do arquivo.

  Os sistemas operacionais subseqüentes, como os sistemas operacionais Unix e Windows, escondem a organização do disco dos usuários e controlam a alocação internamente. Porém, com o tempo, um arquivo seqüencial pode se tornar **fragmentado**; ou seja, seus blocos se tornam espalhados por todo o disco. Para reduzir a fragmentação, o sistema pode fazer uma cópia de backup dos dados no disco e restaurar o disco inteiro. A operação de restauração escreve de volta os blocos de cada ar-

quivo, mas consecutivamente (ou quase isso). Alguns sistemas (como diferentes versões do sistema operacional Windows) possuem utilitários que varrem o disco e depois movem blocos para diminuir a fragmentação. Os aumentos de desempenho observados a partir dessas técnicas podem ser muito grandes.

- **Buffers de escrita não voláteis.** Como o conteúdo da memória principal se perde em casos de falta de energia, as informações sobre atualizações de broadcast precisam ser registradas no disco para que sobrevivam a possíveis falhas do sistema. Por esse motivo, o desempenho de aplicações com uso intensivo de atualização de banco de dados, como sistemas de processamento de transação, depende bastante da velocidade das escritas no disco.

  Podemos usar a **memória de acesso aleatório não volátil** (NVRAM) para agilizar bastante as escritas no disco. O conteúdo da NVRAM não se perde em caso de falta de energia. Um modo comum de implementar a NVRAM é usar uma RAM com bateria. A idéia é que, quando o sistema de banco de dados (ou o sistema operacional) solicitar que um bloco seja gravado no disco, a controladora de disco escreva o bloco no buffer NVRAM e imediatamente notifique ao sistema operacional que a escrita foi completada com sucesso. A controladora grava os dados no seu destino no disco sempre que o disco não tiver quaisquer outras solicitações, ou quando o buffer da NVRAM estiver cheio. Quando o sistema de banco de dados solicitar uma escrita de bloco, ele observa um atraso somente se o buffer da NVRAM estiver cheio. Na recuperação de uma falha do sistema, quaisquer escritas pendentes no buffer da NVRAM são gravadas no disco.

  Buffers de NVRAM são encontrados em certos discos de alto nível, mas são encontrados com mais freqüência em "controladores RAID"; estudamos RAID na próxima seção.

- **Disco de log.** Outra técnica para reduzir as latências de escrita é usar um disco de log – ou seja, um disco dedicado a escrever um log seqüencial – mais ou menos da mesma maneira que um buffer de RAM não volátil. Todo o acesso ao disco de log é seqüencial, basicamente eliminando o tempo de busca, e vários blocos consecutivos podem ser escritos ao mesmo tempo, tornando as escritas no disco de log várias vezes mais rápidas do que as escritas aleatórias. Como antes, os dados também precisam ser escritos em seu local real no disco, mas o disco de log pode fazer a escrita mais tarde, sem que o sistema de banco de dados tenha de esperar até que a escrita termine. Além do mais, o disco de log pode reordenar as escritas para reduzir o movimento do braço do disco. Se o sistema falhar antes do final de algumas escritas no local real do disco, quando o sistema retornar, ele lê o disco de log para descobrir as escritas que não foram completadas, executando-as em seguida.

  Os sistemas de arquivo que admitem discos de log, conforme explicamos, são chamados **sistemas de arquivos journaling**. Os sistemas de arquivos journaling podem ser implementados mesmo sem um disco de log separado, mantendo os dados e o log no mesmo disco. Isso reduz o custo monetário, ao custo de um desempenho inferior.

  A maioria dos sistemas de arquivos modernos implementa o journaling e utiliza o disco de log ao escrever informações internas do sistema de arquivos, como informações de alocação de arquivos. Os sistemas de arquivos mais antigos permitiam a reordenação da escrita sem usar um disco de log, e corriam o risco de corromper as estruturas de dados do sistema de arquivos caso o sistema falhasse. Suponha, por exemplo, que um sistema de arquivo usasse uma lista interligada, e inserisse um novo nó no final, escrevendo primeiro os dados para o novo nó, depois atualizando o ponteiro a partir do nó anterior. Suponha também que as escritas fossem reordenadas, de modo que o ponteiro fosse atualizado primeiro, e o sistema falhasse antes que o novo nó fosse escrito. O conteúdo do nó seria qualquer lixo que estivesse no disco anteriormente, resultando em uma estrutura de dados corrompida.

  Para lidar com a possibilidade de tal problema na estrutura de dados, os sistemas de arquivos da geração anterior tinham de realizar uma verificação de coerência do sistema de arquivos na reinicialização do sistema, para garantir que as estruturas de dados estavam coerentes. E, se não estivessem, outras etapas tinham de ser tomadas para restaurá-las à coerência. Essas verificações resultavam em longos atrasos no reinício do sistema após uma falha, e os atrasos se tornavam piores quando os sistemas de disco aumentavam suas capacidades. Os sistemas de arquivos journaling permitem uma reinicialização mais rápida, sem a necessidade dessas verificações de coerência do sistema de arquivos.

  Porém, as escritas realizadas pelas aplicações normalmente não são escritas no disco de log. Os sistemas de banco de dados implementam suas próprias formas de logging, que estudaremos mais adiante, no Capítulo 17.

RAID

Os requisitos de armazenamento de dados de algumas aplicações (em particular, aplicações Web, banco de dados e multimídia) têm crescido tão rapidamente que um número maior de discos é necessário para armazenar seus dados, embora as capacidades da unidade de disco tenham crescido muito rapidamente.

Ter uma grande quantidade de discos em um sistema apresenta oportunidades para melhorar a velocidade em que os dados podem ser lidos ou escritos, se os discos forem operados em paralelo. Várias leituras ou escritas independentes também podem ser realizadas em paralelo. Além do mais, essa configuração oferece o potencial para melhorar a confiabilidade do armazenamento de dados, pois as informações redundantes podem ser armazenadas em vários discos. Assim, a falha de um disco não leva à perda de dados.

Diversas técnicas de organização de disco, coletivamente chamadas de **Redundant Arrays of Independent Disks (RAID)**, têm sido propostas para alcançar melhor desempenho e confiabilidade.

No passado, os projetistas de sistemas viam os sistemas de armazenamento compostos de vários discos pequenos e baratos como uma alternativa econômica ao uso de discos grandes e caros; o custo por megabyte dos discos menores era menor que o dos discos maiores. De fato, o I de RAID, que agora significa *independent*, originalmente significava *inexpensive* (barato). Hoje, porém, todos os discos são fisicamente pequenos, e os discos de maior capacidade realmente possuem um custo mais baixo por megabyte. Os sistemas RAID são usados por sua maior confiabilidade e maior taxa de desempenho, e não por motivos econômicos. Outra justificação importante para uso do RAID é a maior facilidade de gerenciamento e operações.

Melhoria da confiabilidade por meio da redundância

Vamos considerar primeiro a confiabilidade. A chance de que pelo menos um disco dentre um conjunto de N discos falhe é muito maior do que a chance de que um único disco específico falhe. Suponha que o tempo médio para a falha de um disco seja de 100.000 horas, ou um pouco mais de 11 anos. Então, o tempo médio para a falha de algum disco em um array de 100 discos será 100.000/100 = 1.000 horas, ou cerca de 42 dias, o que não é muito! Se armazenarmos apenas uma cópia dos dados, então cada falha de disco resultará na perda de uma quantidade significativa de dados (conforme discutimos na seção "Características físicas dos discos"). Essa alta freqüência de perda de dados é inaceitável.

A solução para o problema de confiabilidade é introduzir a **redundância**; ou seja, armazenamos informações extras que normalmente não são necessárias, mas que podem ser usadas no caso de falha de um disco, para recriar a informação perdida. Assim, mesmo que um disco falhe, os dados não são pedidos, de modo que o tempo médio efetivo para a falha aumenta, desde que contemos apenas as falhas que levam a perdas de dados ou à não disponibilidade de dados.

A técnica mais simples (porém, mais dispendiosa) de introduzir a redundância é duplicar cada disco. Essa técnica é chamada de **espelhamento** (*mirroring*, ou, às vezes, *shadowing*). Um disco lógico, então, consiste em dois discos físicos, e cada escrita é executada nos dois discos. Se um dos discos falhar, os dados podem ser lidos a partir do outro. Os dados serão perdidos somente se o segundo disco falhar antes que o primeiro disco que falhou seja consertado.

O tempo médio para a falha (ou seja, a perda de dados) de um disco espelhado depende do tempo médio para a falha dos discos individuais, além do **tempo médio para o reparo**, que é o tempo necessário (na média) para substituir um disco que falhou e restaurar seus dados. Suponha que as falhas dos dois discos sejam *independentes*; ou seja, não existe conexão entre a falha de um disco e a falha do outro. Então, se o tempo médio para a falha de um único disco for 100.000 horas, e o tempo médio para o reparo for de 10 horas, então o **tempo médio para a perda de dados** de um sistema de disco espelhado é de $100.000^2/(2 * 10) = 500 * 10^6$ horas, ou 57.000 anos! (Não vamos entrar em mais detalhes aqui; as referências nas notas bibliográficas contêm os detalhes.)

Você precisa estar ciente de que a suposição de independência de falhas de disco não é válida. As faltas de energia e os desastres naturais, como terremotos, incêndios e inundações, podem resultar em danos aos dois discos ao mesmo tempo. Com o envelhecimento dos discos, a probabilidade de falha também aumenta, aumentando a chance de que um segundo disco falhe enquanto o primeiro está sendo reparado. Porém, apesar de todas essas considerações, os sistemas de disco espelhados oferecem muito mais confiabilidade do que os sistemas de único disco. Sistemas de disco espelhados com tempo médio para perda de dados de cerca de 500.000 a 1.000.000 horas, ou 55 a 110 anos, estão disponíveis atualmente.

Faltas de energia são uma fonte de preocupação em particular, pois ocorrem muito mais freqüentemente do que os desastres naturais. As faltas de energia não são um problema se não houver transferência de dados para o disco em andamento quando elas ocorrerem. Porém, mesmo com o espelhamento de discos, se as escritas estiverem ocorrendo no mesmo bloco em ambos os discos, e faltar energia antes que os dois blocos sejam totalmente escritos, os dois podem estar em um estado incoerente. A solução para esse problema é escrever uma cópia primeiro, depois a outra, de modo que uma das duas cópias sempre seja coerente. Algumas ações extras são necessárias quando reinicializamos após uma falta de energia, para recuperar as escritas incompletas. Essa questão é examinada no Exercício prático 11.2.

Melhoria do desempenho por meio do paralelismo

Agora, vamos considerar o benefício do acesso paralelo a vários discos. Com o espelhamento de disco, a velocidade

em que as solicitações de leitura podem ser tratadas é dobrada, de modo que as solicitações de leitura podem ser enviadas a qualquer disco (desde que ambos os discos em um par estejam funcionando, como quase sempre acontece). A taxa de transferência de cada leitura é igual à de um sistema de único disco, mas o número de leituras por unidade de tempo dobrou.

Com vários discos, podemos melhorar a taxa de transferência também (ou em vez de), espalhando dados por vários discos. Em sua forma mais simples, o espalhamento de dados consiste em espalhar os bits de cada byte por vários discos; esse espalhamento é chamado de **espalhamento no nível de bit**. Por exemplo, se tivermos um array de oito discos, escrevemos o bit i de cada byte no disco i. O array de oito discos pode ser tratado como um único disco, com setores que são oito vezes o tamanho normal e, mais importante, que possuem oito vezes a taxa de transferência. Nesse tipo de organização, cada disco participa de cada acesso (leitura ou escrita), de modo que o número de acessos que podem ser processados por segundo é aproximadamente o mesmo que em um único disco, mas cada acesso pode ler oito vezes a quantidade de dados no mesmo tempo que em um único disco. O espalhamento no nível de bit pode ser generalizado a uma quantidade de discos que é um múltiplo de 8 ou um fator de 8. Por exemplo, se usarmos um array de quatro discos, os bits i e $4+i$ de cada byte vão para o disco i.

O **espalhamento no nível de bloco** espalha os blocos por vários discos. Ele trata o array de discos como um único disco grande, e dá aos blocos números lógicos; consideramos que os números de bloco começam com 0. Com um array de n discos, o espalhamento no nível de bloco atribui o bloco lógico i do array de discos ao disco (i mod n) + 1; ele usa o $\lfloor i/n \rfloor^{\text{o}}$ bloco físico do disco para armazenar o bloco lógico i. Por exemplo, com 8 discos, o bloco lógico 0 é armazenado no bloco físico 0 do disco 1, enquanto o bloco lógico 11 é armazenado no bloco físico 1 do disco 4. Ao ler um arquivo grande, o espalhamento no nível de bloco apanha n blocos de cada vez em paralelo, a partir dos n discos, dando uma taxa de transferência de dados alta para leituras grandes. Quando um único bloco é lido, a taxa de transferência de dados é a mesma que em um disco, mas os $n - 1$ discos restantes são livres para realizar outras ações.

O espalhamento no nível de bloco é a forma mais utilizada de espalhamento de dados. Outros níveis de espalhamento, como bytes de um setor ou setores de um bloco, também são possíveis.

Em suma, existem dois objetivos principais para o paralelismo em um sistema de disco:

1. Balancear a carga de vários acessos pequenos (acessos de bloco), de modo que a vazão desses acessos aumente.
2. Realizar grandes acessos em paralelo, de modo que o tempo de resposta dos grandes acessos seja reduzido.

Níveis de RAID

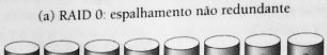
O espelhamento oferece alta confiabilidade, mas é caro. O espalhamento oferece altas taxas de transferência de dados, mas não melhora a confiabilidade. Diversos esquemas alternativos buscam melhorar a redundância com menor custo, combinando o espalhamento de disco com os bits de "paridade" (que descrevemos a seguir). Esses esquemas possuem diferentes opções de custo-desempenho. Os esquemas são classificados em **níveis de RAID**, como na Figura 11.3. (Na figura, P indica bits de correção de erro, e C indica uma segunda cópia dos dados.) Para todos os níveis, a figura representa quatro discos de dados, e os discos extras representados são usados para armazenar informações redundantes para recuperação de falhas.

- **RAID nível 0** oferece aos arrays de disco o espalhamento no nível de blocos, mas sem qualquer redundância (como espelhamento ou bits de paridade). A Figura 11.3a mostra um array de tamanho 4.
- **RAID nível 1** refere-se ao espelhamento de disco com um espalhamento de bloco. A Figura 11.3b mostra uma organização espelhada que mantém quatro discos de dados.

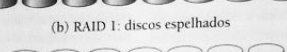
  Observe que alguns fabricantes usam o termo **RAID nível 1+0** ou **RAID nível 10** para se referir ao espelhamento com espalhamento, e utilizam o termo RAID nível 1 para se referir ao espelhamento sem espalhamento. O espelhamento sem espalhamento também pode ser usado com arrays de disco, para dar a aparência de um único disco grande e confiável: se cada disco tiver M blocos, os blocos lógicos de 1 a M são armazenados no disco 1, $M + 1$ até $2M$ no segundo disco, e assim por diante, e cada disco é espelhado.[1]
- **RAID nível 2**, conhecido como organização por código de correção de erro (ECC) no estilo da memória, emprega bits de paridade. Os sistemas de memória há muito tempo têm usado bits de paridade para a detecção e correção de erros. Cada byte em um sistema de memória pode ter um bit de paridade associado a ele, registrando se a quantida-

[1]Observe que alguns fabricantes usam o termo RAID 0+1 para se referir a uma versão de RAID que usa o espalhamento para criar um array RAID 0, e espelha o array para outro array, diferente do RAID 1, no sentido de que, se um disco falhar, o array RAID 0 contendo o disco torna-se inutilizável. O array espelhado ainda pode ser usado, de modo que não existe perda de dados. Esse arranjo é inferior ao RAID 1 quando um disco tiver falhado, pois os outros discos no array RAID 0 podem continuar sendo usados no RAID 1, mas permanecem ociosos no RAID 0+1.

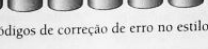
(a) RAID 0: espalhamento não redundante



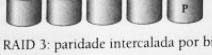
(b) RAID 1: discos espelhados



(c) RAID 2: códigos de correção de erro no estilo da memória



(d) RAID 3: paridade intercalada por bit



(e) RAID 4: paridade intercalada por bloco

(f) RAID 5: paridade distribuída intercalada por bloco

(g) RAID 6: redundância P + Q

Figura 11.3 Níveis de RAID.

de de bits no byte que são definidos como 1 é par (paridade = 0) ou ímpar (paridade = 1). Se um dos bits no byte for modificado (ou 1 se torna 0, ou 0 se torna 1), a paridade do byte muda e, portanto, não combina com a paridade armazenada. De modo semelhante, se o bit de paridade armazenado for danificado, ele não combinará com a paridade calculada. Assim, todos os erros de 1 bit serão detectados pelo sistema de memória. Os esquemas de correção de erro armazenam 2 ou mais bits extras, e podem reconstruir os dados se um único bit for danificado.

A idéia dos códigos de correção de erro pode ser usada diretamente nos arrays de disco, espalhando os bytes pelos discos. Por exemplo, o primeiro bit de cada byte poderia ser armazenado no disco 1, o segundo bit no disco 2, e assim por diante, até que o oitavo bit seja armazenado no disco 8, e os bits de correção de erro sejam armazenados em outros discos.

A Figura 11.3c mostra o esquema de nível 2. Os discos rotulados com *P* armazenam os bits de correção de erro. Se um dos discos falhar, os bits restantes do byte e os bits de correção de erro associados podem ser lidos a partir dos outros discos, e podem ser usados para reconstruir os dados danificados. A Figura 11.3c mostra um array de tamanho 4; observe que RAID nível 2 exige uma sobrecarga de apenas três discos para quatro discos de dados, ao contrário do RAID nível 1, que exigia uma sobrecarga de quatro discos.

- **RAID nível 3**, a organização com paridade intercalada por bit, melhor o nível 2 por explorar o fato de que as controladoras de disco, diferente dos sistemas de memória, podem detectar se um setor foi lido corretamente, de modo que um único bit de paridade pode ser usado para correção de erro, além da detecção. A idéia é a seguinte. Se um dos setores for danificado, o sistema sabe exata-

mente que setor ele é e, para cada bit no setor, o sistema pode descobrir se ele é 1 ou 0, calculando a paridade dos bits correspondentes dos setores nos outros discos. Se a paridade dos bits restantes for igual à paridade armazenada, o bit que falta é 0; caso contrário, ele é 1.

RAID nível 3 é tão bom quanto o nível 2, mas é menos dispendioso no número de discos extras (ele gasta apenas um disco extra), de modo que o nível 2 não é usado na prática. A Figura 11.3d mostra o esquema de nível 3.

RAID nível 3 tem dois benefícios em relação ao nível 1. Ele só precisa de um disco de paridade para vários discos regulares, enquanto o nível 1 precisa de um disco de espelho para cada disco, e, portanto, reduz a sobrecarga de armazenamento. Como as leituras e as escritas de um byte são espalhadas por vários discos, com o espalhamento dos dados em N vias, a taxa de transferência para leitura ou escrita de um único bloco é N vezes mais rápida do que uma organização RAID nível 1 usando o espalhamento com N vias. Por outro lado, RAID nível 3 admite uma quantidade menor de operações de E/S por segundo, pois cada disco precisa participar de cada solicitação de E/S.

- **RAID nível 4**, a organização de paridade intercalada por bloco, utiliza o espalhamento no nível de bloco, assim como RAID 0, e também mantém um bloco de paridade em um disco separado, para os blocos correspondentes de N outros discos. Esse esquema aparece representado na Figura 11.3e. Se um dos discos falhar, o bloco de paridade pode ser usado com os blocos correspondentes a partir de outros discos, para restaurar os blocos do disco que falhou.

  Uma leitura de bloco acessa apenas um disco, permitindo que outras solicitações sejam processadas pelos outros discos. Assim, a taxa de transferência de dados para cada acesso é mais lenta, mas vários acessos de leitura podem prosseguir em paralelo, levando a uma taxa de E/S geral mais alta. As taxas de transferência para grandes leituras são altas, pois todos os discos podem ser lidos em paralelo; grandes escritas também possuem taxas de transferência altas, pois os dados e a paridade podem ser escritos em paralelo.

  Pequenas escritas independentes, por outro lado, não podem ser realizadas em paralelo. Uma escrita de um bloco precisa acessar o disco em que o bloco está armazenado, além do disco de paridade, pois o bloco de paridade precisa ser atualizado. Além do mais, tanto o valor antigo do bloco de paridade quanto o valor antigo do bloco sendo escrito precisam ser lidos para que a nova paridade seja calculada. Assim, uma única escrita exige quatro acessos ao disco: dois para ler os dois blocos antigos, e dois para escrever os dois blocos.
- **RAID nível 5**, a paridade distribuída intercalada por bloco, melhora o nível 4 particionando dados e paridade entre todos os $N + 1$ discos, em vez de armazenar os dados em N discos e a paridade em um disco. No nível 5, todos os discos podem participar satisfazendo solicitações de leitura, ao contrário do RAID nível 4, em que o disco de paridade não pode participar, de modo que o nível 5 aumenta o número total de solicitações que podem ser atendidas em determinada quantidade de tempo. Para cada conjunto de N blocos lógicos, um dos discos armazena a paridade, e os outros N discos armazenam os blocos.

  A Figura 11.3f mostra a configuração. Os Ps são distribuídos por todos os discos. Por exemplo, com um array de 5 discos, o bloco de paridade, rotulado como Pk, para os blocos lógicos $4k$, $4k+1$, $4k+2$, $4k+3$, é armazenado no disco (k mod 5) + 1; os blocos correspondentes dos outros quatro discos armazenam os 4 blocos de dados de $4k$ a $4k+3$. A tabela a seguir indica como os 20 primeiros blocos, numerados de 0 a 19, e seus blocos de paridade, são distribuídos. O padrão mostrado é repetido em outros blocos.

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

Observe que um bloco de paridade não pode armazenar a paridade para blocos no mesmo disco, pois assim uma falha no disco resultaria em perda de dados e também de paridade, e, portanto, não seria recuperável. O nível 5 substitui o nível 4, pois oferece melhor desempenho de leitura-escrita com o mesmo custo, de modo que o nível 4 não é usado na prática.

- **RAID nível 6**, o esquema de redundância P + Q, é muito semelhante ao RAID nível 5, mas armazena informações redundantes extras para proteger contra múltiplas falhas de disco. Em vez de usar paridade, o nível 6 utiliza códigos de correção de erro, como o códigos Reed-Solomon (veja nas notas bibliográficas). No esquema da Figura 11.3g, 2 bits de dados redundantes são armazenados para cada 4 bits de dados – em vez de 1 bit de paridade no nível 5 – e o sistema pode tolerar falhas em dois discos.

Finalmente, observamos que diversas variações foram propostas para os esquemas RAID descritos aqui, e diferentes fornecedores utilizam diferentes terminologias para as variantes.

Escolha do nível de RAID

Os fatores a serem levados em consideração na escolha de um nível RAID são

- Custo monetário dos requisitos extras de armazenamento em disco
- Requisitos de desempenho em termos do número de operações de E/S
- Desempenho quando um disco tiver falhado
- Desempenho durante a reconstrução (ou seja, enquanto os dados em um disco que falhou estão sendo reconstruídos em um novo disco)

O tempo para reconstruir os dados de um disco que falhou pode ser significativo, e varia com o nível RAID que é utilizado. A reconstrução é mais fácil para o RAID nível 1, pois os dados podem ser copiados de outro disco; para os outros níveis, precisamos acessar todos os outros discos no array para reconstruir os dados de um disco que falhou. O **desempenho de reconstrução** de um sistema RAID pode ser um fator importante se a disponibilidade contínua dos dados for um requisito, como em sistemas de banco de dados de alto desempenho. Além do mais, como o tempo de reconstrução pode formar uma parte significativa do tempo de reparo, o desempenho da reconstrução também influencia o tempo médio para a perda de dados.

RAID nível 0 é usado nas aplicações de alto desempenho em que a segurança dos dados não é crítica. Como RAID níveis 2 e 4 foram substituídos por RAID níveis 3 e 5, a escolha de níveis RAID é restrita aos níveis restantes. O espalhamento de bits (nível 3) é inferior ao espalhamento de blocos (nível 5), pois o espalhamento de blocos oferece boas taxas de transferência de dados para grandes transferências, enquanto utiliza menos discos para transferências pequenas. Para transferências pequenas, o tempo de acesso ao disco domina de qualquer forma, de modo que o benefício das leituras paralelas diminui. De fato, o nível 3 pode funcionar pior que o nível 5 para uma transferência pequena, pois a transferência só completa quando os setores correspondentes em todos os discos tiverem sido apanhados; sendo assim, a latência média para o array de disco se torna muito próxima à latência do pior caso para um único disco, invalidando os benefícios de taxas de transferência mais altas. O nível 6 não é aceito atualmente por muitas implementações RAID, mas oferece melhor confiabilidade que o nível 5 e pode ser usado em aplicações em que a segurança dos dados seja muito importante.

A escolha entre RAID nível 1 e nível 5 é mais difícil de se fazer. RAID nível 1 é popular para aplicações como armazenamento de arquivos de log em um sistema de banco de dados, pois oferece o melhor desempenho de escrita. RAID nível 5 tem uma sobrecarga de armazenamento menor que o nível 1, mas possui uma sobrecarga de tempo maior para as escritas. Para aplicações em que os dados são lidos com freqüência e raramente são escritos, o nível 5 é a opção preferida.

Durante muitos anos, as capacidades de armazenamento de disco têm crescido a uma taxa de mais de 50% ao ano, e o custo por byte tem caído na mesma razão. Como resultado, para muitas aplicações de banco de dados existentes, com requisitos de armazenamento moderados, o custo monetário do armazenamento de disco extra necessário ao espelhamento tem se tornado relativamente pequeno (porém, o custo monetário extra continua sendo um problema significativo para aplicações com uso intenso de armazenamento, como o armazenamento de dados de vídeo). As velocidades de acesso têm melhorado em um ritmo muito mais lento (em um fator próximo de 3 durante 10 anos), enquanto o número de operações de E/S exigidas por segundo tem aumentado tremendamente, particularmente para servidores de aplicações Web.

O RAID nível 5, que aumenta o número de operações de E/S necessárias para escrever um único bloco lógico, sofre uma significativa penalidade de tempo em termos de desempenho de escrita. O RAID nível 1, portanto, é o preferido para muitas aplicações com requisitos de armazenamento moderados e altos requisitos de E/S.

Os projetistas de sistemas RAID também precisam tomar muitas outras decisões. Por exemplo, quantos discos deverão haver em um array? Quantos bits deverão ser protegidos por cada bit de paridade? Se houver mais discos em um array, as taxas de transferência de dados são mais altas, mas o sistema seria mais caro. Se houver mais bits protegidos por um bit de paridade, o espaço adicional devido aos bits de paridade é menor, mas existe uma chance maior de que um segundo disco falhe antes que o primeiro disco que falhou seja substituído, e isso resultará em perda de dados.

Questões de hardware

Outra questão na escolha das implementações RAID está no nível do hardware. RAID pode ser implementado sem mudanças no nível do hardware, usando apenas a modificação por software. Essas implementações RAID são denominadas **RAID de software**. Porém, existem benefícios significativos com a construção de um hardware de uso especial para dar suporte ao RAID, conforme esboçamos a seguir; sistemas com suporte especial de hardware são denominados sistemas de **RAID de hardware**.

As implementações RAID de hardware podem utilizar a RAM não volátil para registrar as escritas antes que elas sejam realizadas. No caso de falta de energia, quando o sistema retorna, ele apanha informações sobre quaisquer escritas incompletas a partir da RAM não volátil e depois completa as escritas. Sem esse suporte do hardware, um trabalho extra precisa ser feito para detectar os blocos que podem ter sido escritos parcialmente antes da falta de energia (veja o Exercício prático 11.2).

Algumas implementações RAID de hardware permitem a **troca a quente**; ou seja, os discos com defeito podem ser removidos e substituídos por discos novos sem o desligamento do sistema. A troca a quente (ou *hot swapping*) reduz o tempo médio para o reparo, pois a substituição de um disco não precisa esperar até um momento em que o sistema possa ser desligado. De fato, muitos sistemas críticos hoje executam em uma agenda 24 × 7, ou seja, 24 horas por dia, 7 dias por semana, não oferecendo tempo para desligar e substituir um disco que falhou. Além do mais, muitas implementações RAID atribuem um disco de reserva para cada array (ou para um conjunto de arrays de disco). Se um disco falhar, o disco de reserva é usado imediatamente como um substituto. Como resultado, o tempo médio para o reparo é bastante reduzido, diminuindo as chances de qualquer perda de dados. O disco que falhou pode ser substituído sem pressa.

A fonte de alimentação, ou a controladora de disco, ou ainda a interconexão do sistema em um sistema RAID poderiam se tornar um único ponto de falha, que poderia interromper o funcionamento do sistema RAID. Para evitar essa possibilidade, boas implementações RAID possuem várias fontes de alimentação redundantes (com reservas de bateria, para que continuem a funcionar mesmo que falte energia). Esses sistemas RAID possuem diversas interfaces de disco e diversas interconexões para conectar o sistema RAID ao sistema de computador (ou a uma rede de sistemas de computador). Assim, a falha de qualquer componente isolado não interromperá o funcionamento do sistema RAID.

Outras aplicações de RAID

Os conceitos de RAID têm sido generalizados para outros dispositivos de armazenamento, como os arrays de fitas, e até mesmo para o broadcast de dados por sistemas sem fio. Quando são aplicadas a arrays de fitas, as estruturas RAID são capazes de recuperar dados mesmo que uma das fitas em um array de fitas seja danificada. Quando aplicadas ao broadcast de dados, um bloco de dados é dividido em unidades pequenas e enviado junto com uma unidade de paridade; se uma das unidades não for recebida por qualquer motivo, ela pode ser reconstruída a partir das outras unidades.

Armazenamento terciário

Em um grande sistema de banco de dados, alguns dos dados podem ter de residir no armazenamento terciário. Os dois meios de armazenamento terciário mais comuns são discos ópticos e fitas magnéticas.

Discos ópticos

Os compact disks têm sido um meio popular para distribuir software, dados de multimídia como áudio e imagens e outras informações publicadas eletronicamente. Eles possuem uma capacidade muito grande (640 megabytes) e são baratos para produção em massa. Os digital video disks (DVDs) agora estão substituindo os compact disks nas aplicações que exigem quantidades maiores de dados. Os discos no formato DVD-5 podem armazenar 4,7 gigabytes de dados (em uma camada de gravação), enquanto os discos no formato DVD-9 podem armazenar 8,5 gigabytes de dados (em duas camadas de gravação). A gravação nos dois lados de um disco gera capacidades ainda maiores; os formatos DVD-10 e DVD-18, que são as versões em dois lados do DVD-5 e DVD-9, podem armazenar 9,4 gigabytes e 17 gigabytes, respectivamente. Formatos mais recentes, chamados de *HD-DVD* e *DVD Blu-Ray*, possuem uma capacidade significativamente maior: discos HD-DVD podem armazenar de 12 a 30 gigabytes por disco, enquanto os discos Blu-Ray DVD podem armazenar de 25 a 50 gigabytes por disco. Estes deverão ser bastante utilizados por volta de 2006-2007.

Unidades de CD e DVD possuem tempos de busca muito maiores (100 milissegundos é comum) do que as unidades de disco magnético, pois a montagem da cabeça de acesso é mais pesada. As velocidades rotacionais normalmente são menores do que as dos discos magnéticos, embora as unidades de CD e DVD mais rápidas tenham velocidades de rotação de aproximadamente 3000 rotações por minuto, que é comparável às velocidades das unidades de disco magnéticos mais inferiores. As velocidades rotacionais das unidades de CD originalmente correspondem aos padrões do CD de áudio, e as velocidades das unidades de DVD originalmente correspondem aos padrões do DVD de vídeo, mas as unidades da geração atual giram muitas vezes mais do que na velocidade padrão.

As taxas de transferência de dados são um pouco menores do que para os discos magnéticos. As unidades atuais de CD lêem em torno de 3 a 6 megabytes por segundo, e as unidades de DVD lêem 8 a 20 megabytes por segundo. Assim como as unidades de disco magnético, os discos ópticos armazenam mais dados nas trilhas externas e menos dados nas trilhas internas. A taxa de transferência das unidades ópticas é caracterizada como *nx*, o que significa que a unidade admite transferências em *n* vezes a taxa padrão; velocidades de cerca de 50x para CD e 16x para DVD agora são comuns.

As versões dos discos ópticos para única escrita (CD-R, DVD-R e DVD+R) são populares para distribuição de dados e particularmente para arquivamento de dados, pois possuem alta capacidade, tempo de vida maior do que os discos magnéticos e podem ser removidas e armazenadas em um local remoto. Como não podem ser regravadas, elas podem ser usadas para armazenar informações que não deverão ser modificadas, como trilhas de auditoria. As versões para múl-

tiplas escritas (CD-RW, DVD-RW, DVD+RW e DVD-RAM) também são usadas para fins de arquivamento.

Jukeboxes são dispositivos que armazenam uma grande quantidade de discos ópticos (até várias centenas) e os carregam automaticamente por demanda em uma dentre um pequeno número de unidades (normalmente, de 1 a 10). A capacidade de armazenamento agregado de tal sistema pode ser de muitos terabytes. Quando um disco é acessado, ele é carregado por um braço mecânico de um rack para uma unidade (qualquer disco que já estivesse na unidade primeiro precisa ser colocado de volta ao rack). O tempo para carregar/descarregar o disco normalmente é de alguns segundos – muito maior do que os tempos de acesso ao disco.

Fitas magnéticas

Embora as fitas magnéticas sejam relativamente permanentes, podendo manter grandes volumes de dados, elas são lentas em comparação com os discos magnéticos e ópticos. Ainda mais importante, as fitas magnéticas são limitadas ao acesso seqüencial. Assim, elas não podem oferecer o acesso aleatório para requisitos de armazenamento secundário, embora historicamente, antes do uso dos discos magnéticos, as fitas fossem usadas como meio de armazenamento secundário.

As fitas são usadas principalmente para backup, para armazenamento de informações usadas com pouca freqüência e como meio offline para transferir informações de um sistema para outro. As fitas também são usadas para armazenar grandes volumes de dados, como dados de vídeo ou imagem, que não precisam ser acessíveis rapidamente ou que são tão volumosos que o armazenamento em disco magnético seria muito dispendioso.

Uma fita é mantida em um spool e é bobinada e rebobinada para passar por um cabeçote de leitura-escrita. A movimentação para o ponto correto em uma fita pode levar alguns segundos, ou mesmo minutos, em vez de milissegundos; porém, uma vez posicionadas, as unidades de fita podem gravar dados em densidades e velocidades que se aproximam as das unidades de disco. As capacidades variam, dependendo da extensão e largura da fita e da densidade em que o cabeçote pode ler e gravar. O mercado atualmente está fragmentado entre uma grande variedade de formatos de fita. As capacidades de fita atualmente disponíveis variam de alguns gigabytes com o formato **Digital Audio Tape** (DAT), 10 a 40 gigabytes com o formato **Digital Linear Tape** (DLT), 100 gigabytes e mais com o formato **Ultrium**, até 330 gigabytes com formatos de fita **Ampex helical scan**. As taxas de transferência de dados são da ordem de alguns até dezenas de megabytes por segundo.

Unidades de fita são bastante confiáveis, e bons sistemas de unidade de fita realizam uma leitura dos dados recém-gravados para garantir que eles foram registrados corretamente. Porém, as fitas possuem limites no número de vezes que podem ser lidas ou gravadas de modo confiável.

Os **jukeboxes de fita**, assim como os jukeboxes ópticos, mantêm grandes quantidades de fitas, com algumas unidades em que as fitas podem ser montadas; eles são usados para armazenar grandes volumes de dados, chegando até a muitos petabytes (10^{15} bytes), com tempos de acesso na ordem de segundos até alguns minutos. As aplicações que precisam dessa grande quantidade de armazenamento de dados incluem sistemas de imagens que reúnem dados gerados por satélites remotos e grandes bibliotecas de vídeo para operadoras de televisão.

Alguns formatos de fita (como o formato Accelis) admitem tempos de busca ainda mais rápidos (da ordem de dezenas de segundos) e servem para aplicações que recuperam informações de jukeboxes. A maioria dos outros formatos de fita oferece capacidades maiores, ao custo de um acesso mais lento; esses formatos são ideais para backup de dados, em que as buscas rápidas não são importantes.

As unidades de fita não conseguiram acompanhar as enormes melhorias na capacidade da unidade de disco e a correspondente redução no custo de armazenamento. Embora o custo das fitas seja baixo, o custo das unidades de fita e bibliotecas de fita é muito mais alto do que o custo de uma unidade de disco: uma biblioteca de fita capaz de armazenar alguns terabytes pode custar dezenas de milhares de dólares. O backup de dados em unidades de disco tornou-se uma alternativa econômica ao backup de fita para diversas aplicações.

Acesso ao armazenamento

Um banco de dados é mapeado em uma série de arquivos diferentes, que são mantidos pelo sistema operacional em uso. Esses arquivos residem permanentemente em discos, com backups em fitas. Cada arquivo é particionado em unidades de armazenamento de tamanho fixo, chamadas **blocos**, que são as unidades de alocação de armazenamento e transferência de dados. Na seção "Organização de arquivo", discutiremos várias maneiras de organizar os dados logicamente em arquivos.

Um bloco pode conter vários itens de dados. O conjunto exato de itens de dados que um bloco contém é determinado pelo formato da organização física dos dados sendo usados (ver seção "Organização de arquivo"). Consideraremos que nenhum item de dados se espalha por dois ou mais blocos. Essa suposição é realista para a maioria das aplicações de processamento de dados, como nosso exemplo bancário.

Um objetivo importante do sistema de banco de dados é minimizar o número de transferências de bloco entre o disco e a memória. Um modo de reduzir a quantidade de aces-

sos ao disco é manter o máximo de blocos possível na memória principal. O objetivo é aumentar as chances de que, quando um bloco for acessado, ele já esteja na memória e, assim, nenhum acesso ao disco seja exigido.

Como não é possível manter todos os blocos na memória principal, temos de gerenciar a alocação do espaço disponível na memória principal para o armazenamento de blocos. O **buffer** é aquela parte da memória principal disponível para armazenamento de cópias de blocos de disco. Há sempre uma cópia de cada bloco mantida no disco, mas a cópia no disco pode ser uma versão do bloco mais antiga que a versão no buffer. O subsistema responsável pela alocação do espaço no buffer é chamado de **gerenciador de buffer**.

Gerenciador de buffer

Os programas em um sistema de banco de dados fazem solicitações (ou seja, chamadas) ao gerenciador de buffer quando precisam de um bloco do disco. Se o bloco já estiver no buffer, o gerenciador de buffer passa o endereço do bloco na memória principal para o solicitante. Se o bloco não estiver no buffer, o gerenciador de buffer primeiro aloca espaço no buffer para o bloco, eliminando algum outro bloco, se for necessário, para criar espaço para o novo bloco. O bloco descartado é gravado de volta no disco somente se tiver sido modificado desde a hora mais recente em que foi gravado no disco. Depois, o gerenciador de buffer lê o bloco solicitado do disco para o buffer, e passa o endereço do bloco na memória principal ao solicitante. As ações internas do gerenciador de buffer são transparentes aos programas que emitem solicitações de bloco de disco.

Se você estiver acostumado com os conceitos do sistema operacional, notará que o gerenciador de buffer parece ser nada mais do que um gerenciador de memória virtual, como aqueles encontrados na maioria dos sistemas operacionais. Uma diferença é que o tamanho do banco de dados pode ser muito mais do que o espaço de endereço de hardware de uma máquina, de modo que os endereços de memória não são suficientes para endereçar todos os blocos de disco. Além disso, para atender bem ao sistema de banco de dados, o gerenciador de buffer precisa usar técnicas mais sofisticadas do que os esquemas típicos de gerenciamento de memória virtual:

- **Estratégia de substituição de buffer.** Quando não há espaço no buffer, um bloco precisa ser removido do buffer antes que um novo possa ser lido. A maioria dos sistemas operacionais utiliza um esquema de uso menos recente (**LRU – Least Recently Used**), em que o bloco que foi referenciado menos recentemente é gravado no disco e removido do buffer. A técnica simples pode ser melhorada para aplicações de banco de dados.
- **Blocos presos.** Para o sistema de banco de dados ser capaz de se recuperar de falhas (Capítulo 17), é necessário restringir aqueles momentos em que um bloco pode ser gravado no disco. Por exemplo, a maior parte dos sistemas de recuperação exige que um bloco não deve ser gravado no disco enquanto uma atualização no bloco está em andamento. Um bloco que não tem permissão para ser gravado no disco é considerado **preso**. Embora muitos sistemas operacionais não aceitem blocos presos, esse recurso é essencial para um sistema de banco de dados tolerante a falhas.
- **Saída forçada de blocos.** Existem situações em que é necessário gravar o bloco de volta no disco, embora o espaço no buffer que ele ocupa não seja necessário. Essa escrita é chamada de **saída forçada** de um bloco. Veremos o motivo para a saída forçada no Capítulo 17; resumindo, o conteúdo da memória principal e, portanto, o conteúdo do buffer, se perde em uma falha, enquanto os dados no disco normalmente sobrevivem a uma falha.

Políticas de substituição de buffer

O objetivo de uma estratégia de substituição para blocos no buffer é minimizar os acessos ao disco. Para programas de uso geral, não é possível prever com precisão quais blocos serão referenciados. Portanto, os sistemas operacionais usam o padrão passado de referências ao bloco como previsão para referências futuras. Geralmente se supõe que os blocos que foram referenciados recentemente provavelmente serão referenciados de novo. Portanto, se um bloco tiver de ser substituído, o bloco referenciado menos recentemente será substituído. Essa técnica é chamada de esquema de substituição de bloco **usado menos recentemente** (**LRU – Least Recently Used**).

LRU é um esquema de substituição aceitável nos sistemas operacionais. Porém, um sistema de banco de dados é capaz de prever o padrão de referências futuras com mais precisão do que um sistema operacional. Uma solicitação do usuário ao sistema de banco de dados envolve várias etapas. O sistema de banco de dados normalmente é capaz de determinar com antecedência quais blocos serão necessários examinando cada uma das etapas exigidas para realizar a operação solicitada pelo usuário. Assim, diferente dos sistemas operacionais, que precisam contar com o passado para prever o futuro, os sistemas de banco de dados podem ter informações considerando pelo menos o futuro a curto prazo. Para ilustrar como as informações sobre o acesso futuro ao bloco nos permitem melhorar a estratégia LRU, considere o processamento da expressão da álgebra relacional.

```
for each tuple b of credor do
  for each tuple c of cliente do
    if b[nome-cliente] = c[nome-cliente]
    then begin
           let x be a tuple defined as follows:
           x[nome-cliente] := b[nome-cliente]
           x[número-empréstimo] := b[número-empréstimo]
           x[rua-cliente] := c[rua-cliente]
           x[cidade-cliente] := c[cidade-cliente]
           incluir tupla x como parte do resultado de credor ⋈ cliente
         end
  end
end
```

Figura 11.4 Procedimento para calcular a junção.

Considere que a estratégia escolhida para processar essa solicitação seja dada pelo programa em pseudocódigo mostrado na Figura 11.4. (Estudaremos outras estratégias mais eficientes no Capítulo 13.)

Considere que as duas relações deste exemplo estejam armazenadas em arquivos separados. Neste exemplo, podemos ver que, quando uma tupla de *credor* tiver sido processada, essa tupla não será necessária novamente. Portanto, quando o processamento de um bloco inteiro de tuplas *credor* estiver concluído, esse bloco não será mais necessário na memória, embora possa ter sido usado recentemente. O gerenciador de buffer deve ser instruído a liberar o espaço ocupado por um bloco *credor* assim que a tupla final tiver sido processada. Essa estratégia de gerenciamento de buffer é denominada estratégia de **lançar imediatamente**.

Agora, considere os blocos contendo tuplas de *cliente*. Precisamos examinar cada bloco de tuplas de *cliente* uma vez para cada tupla da relação *credor*. Quando o processamento do bloco de um *cliente* terminar, sabemos que esse bloco não será acessado novamente até que todos os outros blocos de *cliente* tenham sido processados. Assim, o bloco de *cliente* usado mais recentemente será o bloco final a ser referenciado de novo, e o bloco de *cliente* usado menos recentemente é o bloco que será referenciado em seguida. Essa suposição é exatamente o oposto daquela que forma a base para a estratégia LRU. Na realidade, a estratégia ideal para a substituição de bloco para o procedimento anterior é a estratégia **usado mais recentemente (MRU – Most Recently Used)**. Se o bloco de um *cliente* tiver de ser removido do buffer, a estratégia MRU escolhe o bloco usado mais recentemente (os blocos não são elegíveis para substituição enquanto estão sendo usados).

Para que a estratégia MRU funcione de forma correta para o nosso exemplo, o sistema precisa prender o bloco de *cliente* atualmente sendo processado. Depois que a última tupla de *cliente* tiver sido processada, o bloco será liberado, e se tornará o bloco usado mais recentemente.

Além de usar o conhecimento que o sistema pode ter a respeito da solicitação sendo processada, o gerenciador de buffer pode usar informações estatísticas sobre a probabilidade de que uma solicitação referencie uma relação em particular. Por exemplo, o dicionário de dados que (como veremos em detalhes na seção "Armazenamento em dicionário de dados") acompanha o esquema lógico das relações e também sua informação de armazenamento físico é uma das partes do banco de dados acessadas com mais freqüência. Assim, o gerenciador de buffer deverá tentar não remover os blocos do dicionário de dados da memória principal, a menos que outros fatores exijam que ele faça isso. No Capítulo 12, discutimos sobre índices de arquivos. Como um índice para um arquivo pode ser acessado com mais freqüência do que o próprio arquivo, em geral, o gerenciador de buffer não deverá remover os blocos de índice da memória principal se houver alternativas.

A estratégia ideal para substituição de blocos do banco de dados precisa de conhecimento das operações do banco de dados – tanto aquelas sendo realizadas quanto aquelas que serão realizadas no futuro. Não se conhece qualquer estratégia isolada para lidar bem com todos os cenários possíveis. Na realidade, um número surpreendentemente grande de sistemas de banco de dados utiliza LRU, apesar das falhas dessa estratégia. As perguntas e os exercícios práticos exploram estratégias alternativas.

A estratégia que o gerenciador de buffer utiliza para substituição de bloco é influenciada por fatores diferentes que o momento em que o bloco será referenciado novamente. Se o sistema estiver processando solicitações por vários usuários ao mesmo tempo, o subsistema de controle de concorrência (Capítulo 16) pode ter de adiar certas solicitações, para garantir a preservação da coerência do banco de dados. Se o gerenciador de buffer receber informações do subsistema de controle de concorrência, indicando quais solicitações estão sendo adiadas, ele pode usar essa informação para alterar sua estratégia de substituição de

bloco. Especificamente, os blocos necessários para as solicitações ativas (não adiadas) podem ser retidos no buffer às custas dos blocos necessários pelas solicitações adiadas.

O subsistema de recuperação de falhas (Capítulo 17) impõe fortes restrições sobre a substituição de blocos. Se um bloco tiver sido modificado, o gerenciador de buffer não tem permissão para gravar a nova versão do bloco do buffer para o disco, pois isso destruiria a versão antiga. Em vez disso, o gerenciador de bloco precisa buscar permissão do subsistema de recuperação de falhas antes de gravar um bloco. O subsistema de recuperação de falhas pode exigir que certos outros blocos sejam forçados a sair antes de conceder permissão para o gerenciador de buffer enviar o bloco solicitado. No Capítulo 17, definimos exatamente a interação entre o gerenciador de buffer e o subsistema de recuperação de falhas.

Organização de arquivo

Um arquivo é organizado logicamente como uma seqüência de registros. Esses registros são mapeados em blocos de disco. Os arquivos são fornecidos como uma construção básica nos sistemas operacionais, de modo que assumiremos a existência de um *sistema de arquivos* básico. Precisamos considerar as formas de representar os modelos de dados lógicos em termos de arquivos.

Embora os blocos sejam de um tamanho fixo, determinado pelas propriedades físicas do disco e pelo sistema operacional, os tamanhos de registro variam. Em um banco de dados relacional, tuplas de relações distintas geralmente são de tamanhos diferentes.

Uma técnica para mapear o banco de dados em arquivos é usar vários arquivos e armazenar registros de apenas um tamanho fixo em qualquer arquivo determinado. Uma alternativa é estruturar nossos arquivos de modo que possamos acomodar vários tamanhos de registros; porém, arquivos com registros de tamanho fixo são mais fáceis de implementar do que os arquivos com registros de tamanho variável. Muitas das técnicas utilizadas para o primeiro tipo podem ser aplicadas ao caso do tamanho variável. Assim, começamos considerando um arquivo com registros de tamanho fixo.

Registros de tamanho fixo

Como um exemplo, vamos considerar um arquivo de registros de conta para nosso banco de dados bancário. Cada registro desse arquivo é definido (em pseudocódigo) como:

```
type depósito = record
                    número_conta char(10);
                    nome_agência char(22);
                    saldo numeric(12,2);
                end
```

Se considerarmos que cada caractere ocupa 1 byte e que numeric(12,2) ocupa 8 bytes, nosso registro *conta* possui 40 bytes de extensão. Uma técnica simples é usar os 40 primeiros bytes para o primeiro registro e os 40 bytes seguintes para o segundo registro, e assim por diante (Figura 11.5). Porém, existem dois problemas com essa técnica simples:

1. É difícil excluir um registro a partir dessa estrutura. O espaço ocupado pelo registro a ser excluído precisa ser preenchido com algum outro registro do arquivo, ou precisamos ter um meio de marcar os registros excluídos de modo que possam ser ignorados.
2. A menos que o tamanho do bloco seja um múltiplo de 40 (o que é improvável), alguns registros cruzarão os limites do bloco. Ou seja, parte do registro será armazenada em um bloco e parte em outro. Assim, serão necessários dois acessos de bloco para ler ou escrever tal registro.

Quando um registro é excluído, poderíamos mover o registro que veio depois dele para o espaço ocupado anteriormente pelo registro excluído, e assim por diante, até que

registro 0	A-102	Perryridge	400
registro 1	A-305	Round Hill	350
registro 2	A-215	Mianus	700
registro 3	A-101	Downtown	500
registro 4	A-222	Redwood	700
registro 5	A-201	Perryridge	900
registro 6	A-217	Brighton	750
registro 7	A-110	Downtown	600
registro 8	A-218	Perryridge	700

Figura 11.5 Arquivo contendo registros de *conta*.

registro 0	A-102	Perryridge	400
registro 1	A-305	Round Hill	350
registro 3	A-101	Downtown	500
registro 4	A-222	Redwood	700
registro 5	A-201	Perryridge	900
registro 6	A-217	Brighton	750
registro 7	A-110	Downtown	600
registro 8	A-218	Perryridge	700

Figura 11.6 Arquivo da Figura 11.5 com o registro 2 excluído e todos os registros movidos.

cada registro após o registro excluído tenha sido movido para a frente (Figura 11.6). Essa técnica exige mover uma grande quantidade de registros. Poderia ser mais fácil simplesmente mover o registro final do arquivo para o espaço ocupado pelo registro excluído (Figura 11.7).

Não é desejável mover registros para ocupar o espaço liberado por um registro excluído, pois isso exige acessos adicionais ao bloco. Como as inserções costumam ser mais freqüentes do que as exclusões, é aceitável deixar aberto o espaço ocupado pelo registro excluído e esperar por uma inserção subseqüente antes de reutilizar o espaço. Um marcador simples em um registro excluído não é suficiente, pois é difícil encontrar esse espaço disponível quando uma inserção está sendo feita. Assim, precisamos introduzir uma estrutura adicional.

No início do arquivo, alocamos um certo número de bytes como um **cabeçalho de arquivo**. O cabeçalho terá uma série de informações sobre o arquivo. Por enquanto, tudo o que precisamos armazenar lá é o endereço do segundo registro disponível, e assim por diante. Intuitivamente, podemos pensar nesses endereços armazenados como *ponteiros*, pois eles apontam para o local de um registro. Os registros excluídos, assim, formam uma lista interligada, que normalmente é conhecida como **lista livre**. A Figura 11.8 mostra o arquivo da Figura 11.5, com a lista livre, após os registros 1, 4 e 6 terem sido excluídos.

Na inserção de um novo registro, usamos o registro apontado pelo cabeçalho. Mudamos o ponteiro do cabeçalho para que aponte para o próximo registro disponível. Se não houver espaço disponível, acrescentamos o novo registro ao final do arquivo.

A inserção e a exclusão para arquivos de registros com tamanho fixo são simples de implementar, pois o espaço disponível por um registro excluído é exatamente o espaço necessário para inserir um registro. Se permitirmos registros com tamanho variável em um arquivo, essa combinação não permanece. Um registro inserido pode não caber no espaço que ficou livre por um registro excluído, ou pode caber apenas parcialmente nesse espaço.

Registros de tamanho variável

Registros com tamanho variável surgem em sistemas de banco de dados de várias maneiras:

- Armazenamento de vários tipos de registro em um arquivo
- Tipos de registro que permitem tamanhos variáveis para um ou mais campos
- Tipos de registro que permitem campos repetidos, como arrays ou multiconjuntos

Existem diferentes técnicas para implementar registros de tamanho variável.

registro 0	A-102	Perryridge	400
registro 1	A-305	Round Hill	350
registro 8	A-218	Perryridge	700
registro 3	A-101	Downtown	500
registro 4	A-222	Redwood	700
registro 5	A-201	Perryridge	900
registro 6	A-217	Brighton	750
registro 7	A-110	Downtown	600

Figura 11.7 Arquivo da Figura 11.5 com o registro 2 excluído e o registro final movido.

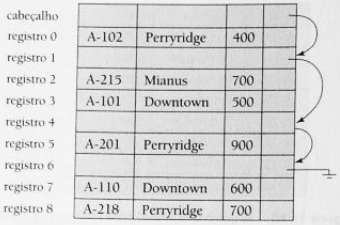


Figura 11.8 Arquivo da Figura 11.5, com a lista livre após a exclusão dos registros 1, 4 e 6.

A **estrutura de página em slot** normalmente é usada para organizar registros dentro de um bloco, e aparece na Figura 11.9. Existe um cabeçalho no início de cada bloco, contendo a seguinte informação:

1. O número de entradas de registro no cabeçalho
2. O final do espaço livre no bloco
3. Um array cujas entradas contêm o local e o tamanho de cada registro

Os registros reais são alocados de forma *contígua* no bloco, começando do final do bloco. O espaço livre no bloco é contíguo, entre a entrada final no array do cabeçalho e o primeiro registro. Se um registro for inserido, o espaço é alocado para ele no final do espaço livre, e uma entrada contendo seu tamanho e local é acrescentada ao cabeçalho.

Se um registro for excluído, o espaço que ele ocupa é liberado, e sua entrada é definida para ser excluída (seu tamanho é definido como –1, por exemplo). Além disso, os registros no bloco antes do registro excluído são movidos, de modo que o espaço livre criado pela exclusão é ocupado, e todo o espaço livre novamente fica entre a entrada final no array do cabeçalho e o primeiro registro. O ponteiro de fim do espaço livre no cabeçalho também é atualizado corretamente. Os registros podem aumentar ou diminuir por meio de técnicas semelhantes, desde que haja espaço no bloco. O custo para mover os registros não é muito alto, pois o tamanho de um bloco é limitado: um valor típico é de 4 kilobytes.

A estrutura de página em slot exige que não haja ponteiros que apontem diretamente para os registros. Em vez disso, os ponteiros precisam apontar para a entrada no cabeçalho que contém o local real do registro. Esse nível de indireção permite que os registros sejam movidos para evitar fragmentação de espaço dentro de um bloco, enquanto admite ponteiros indiretos para o registro.

Os bancos de dados normalmente armazenam dados que podem ser muito maiores do que um bloco de disco. Por exemplo, uma imagem ou uma gravação de áudio podem ter vários megabytes, enquanto um objeto de vídeo pode ocupar vários gigabytes. Lembre-se de que a SQL admite os tipos **blob** e **clob**, que armazenam objetos binários e de caractere muito grandes.

A maioria dos bancos de dados relacionais restringe o tamanho de um registro para não ser maior que o tamanho de um bloco, de modo a simplificar o gerenciamento de buffer e o gerenciamento de espaço livre. Objetos grandes normalmente são armazenados em um arquivo especial (ou coleção de arquivos) em vez de serem armazenados com os outros atributos (curtos) dos registros em que ocorrem.

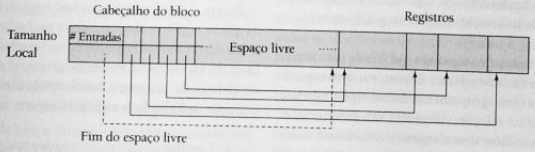


Figura 11.9 Estrutura de página em slots.

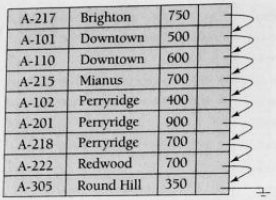


Figura 11.10 Arquivo seqüencial para registros de *conta*.

Objetos grandes normalmente são representados por meio de organizações de arquivo em árvore B+, que estudaremos na seção "Organização de arquivos de árvore B+" do Capítulo 12. Organizações de arquivo em árvore B+ permitem ler um objeto inteiro, ou intervalos de bytes especificados no objeto, além de inserir e excluir partes do objeto.

Organização de registros em arquivos

Até aqui, estudamos como os registros são representados em uma estrutura de arquivos. Uma relação é um conjunto de registros. Dado um conjunto de registros, a próxima pergunta é como organizá-los em um arquivo. Várias das possíveis maneiras de organizar registros em arquivos são:

- **Organização de arquivos em heap.** Qualquer registro pode ser colocado em qualquer lugar no arquivo onde existe espaço para o registro. Não existe ordenação de registros. Normalmente, existe um único arquivo para cada relação.
- **Organização seqüencial de arquivos.** Os registros são armazenados em ordem seqüencial, de acordo com o valor de uma "chave de busca" de cada registro. A próxima seção descreve essa organização.
- **Organização de arquivos com hashing.** Uma função de hash é calculada sobre algum atributo de cada registro. O resultado da função de hash especifica em que bloco do arquivo o registro deve ser colocado. O Capítulo 12 descreve essa organização; ela está bastante relacionada às estruturas de indexação descrita naquele capítulo.

Geralmente, um arquivo separado é utilizado para armazenar os registros de cada relação. Porém, em uma **organização de arquivos com agrupamento de múltiplas tabelas**, os registros de várias relações diferentes são armazenados no mesmo arquivo; além disso, registros relacionados das diferentes relações são armazenados no mesmo bloco, de modo que uma operação de E/S busca registros relacionados de todas as relações. Por exemplo, os registros das duas relações podem ser considerados como relacionados se combinarem em uma junção das duas relações. A seção "Organização de arquivos com agrupamento de múltiplas tabelas" descreve essa organização.

Organização seqüencial de arquivos

Um **arquivo seqüencial** é projetado para o processamento eficiente de registros em ordem, com base em alguma chave de busca. Uma **chave de busca** é qualquer atributo ou conjunto de atributos; ela não precisa ser a chave primária, ou mesmo uma superchave. Para permitir a rápida recuperação de registros na ordem da chave de busca, encadeamos os registros pelos ponteiros. O ponteiro em cada registro aponta para o próximo registro na ordem da chave de busca. Além do mais, para reduzir o número de acessos de bloco no processamento seqüencial de arquivos, armazenamos registros fisicamente em ordem de chave de busca, ou o mais próximo possível da ordem da chave de busca.

A Figura 11.10 mostra um arquivo seqüencial de registros de *conta* tomados do nosso exemplo bancário. Nesse exemplo, os registros são armazenados na ordem da chave de busca, usando *nome-agência* como chave de busca.

A organização de arquivo seqüencial permite que os registros sejam lidos na ordem classificada; isso pode ser útil para fins de exibição, além de certos algoritmos de processamento de consulta que estudaremos no Capítulo 13.

Porém, é difícil manter a ordem seqüencial física à medida que os registros são inseridos e excluídos, pois é dispendioso mover muitos registros como resultado de uma única inserção ou exclusão. Podemos gerenciar a exclusão usando cadeias de ponteiros, como vimos anteriormente. Por exemplo, aplicamos as seguintes regras:

1. Localize o registro no arquivo que vem antes do registro a ser inserido na ordem da chave de busca.

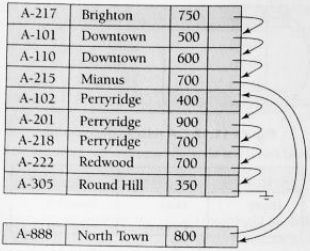


Figura 11.11 Arquivo seqüencial após uma inserção.

2. Se houver um registro livre (ou seja, espaço criado após uma exclusão) dentro do mesmo bloco desse registro, insira o novo registro lá. Caso contrário, insira o novo registro em um *bloco de estouro*. Nos dois casos, ajuste os ponteiros de modo a encadear os registros na ordem da chave de busca.

A Figura 11.11 mostra o arquivo da Figura 11.10 após a inserção do registro (North Town, A-888, 800). A estrutura da Figura 11.11 permite a inserção rápida de novos registros, mas força as aplicações de processamento seqüencial de arquivos a processarem registros em uma ordem que não combina com a ordem física dos registros.

Se relativamente poucos registros tiverem de ser armazenados em blocos de estouro, essa técnica funcionará bem. Porém, em alguns casos, a correspondência entre a ordem da chave de busca e a ordem física pode ser totalmente perdida, e o processamento seqüencial se tornará muito menos eficiente. Nesse ponto, o arquivo deverá ser **reorganizado** de modo que mais uma vez esteja fisicamente na ordem seqüencial. Essas reorganizações são dispendiosas, e precisam ser feitas durante momentos em que a carga do sistema é baixa. A freqüência com que as reorganizações são necessárias depende da freqüência de inserção de novos registros. Em um caso extremo em que as inserções raramente ocorrem, sempre é possível manter o arquivo na ordem fisicamente classificada. Nesse caso, o campo de ponteiro da Figura 11.10 não é necessário.

Organização de arquivos com agrupamento de múltiplas tabelas

Muitos sistemas de banco de dados relacionais armazenam cada relação em um arquivo separado, de modo que podem tirar todo o proveito do sistema de arquivos que o sistema operacional oferece. Normalmente, as tuplas de uma relação podem ser representadas como registros de tamanho fixo. Assim, as relações podem ser mapeadas para uma estrutura de arquivo simples. Essa implementação simples de um sistema de banco de dados relacional é bem adequada às implementações de banco de dados de baixo custo como, por exemplo, em sistemas embutidos ou dispositivos portáteis. Nesses sistemas, o tamanho do banco de dados é pequeno, de modo que pouca coisa é ganha com uma estrutura de arquivos sofisticada. Além do mais, em tais ambientes, é essencial que o tamanho geral do código do objeto para o sistema de banco de dados seja pequeno. Uma estrutura de arquivos simples reduz a quantidade de código necessária para implementar o sistema.

Essa técnica simples para a implementação de banco de dados relacional torna-se menos satisfatória quando o tamanho do banco de dados aumenta. Já vimos que existem vantagens no desempenho que são ganhas com a atribuição cuidadosa de registros aos blocos e com a organização cuidadosa dos próprios blocos. Certamente, uma estrutura de arquivos mais complicada pode ser benéfica, mesmo se retivermos a estratégia de armazenar cada relação em um arquivo separado.

Porém, muitos sistemas de banco de dados em grande escala não contam diretamente com o sistema operacional básico para o gerenciamento de arquivos. Em vez disso, um arquivo grande do sistema operacional é alocado ao sistema de banco de dados. O sistema de banco de dados armazena todas as relações nesse único arquivo e gerencia o próprio arquivo. Para ver a vantagem de armazenar muitas relações em um arquivo, considere a seguinte consulta SQL para o banco de dados bancário:

select *número-conta, nome-cliente, rua-cliente, cidade-cliente*
from *depositante, cliente*
where *depositante.nome-cliente = cliente.nome-cliente* }

nome_cliente	*numero_conta*
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

Figura 11.12 A relação *depositante*.

nome_cliente	*rua_cliente*	*cidade_cliente*
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

Figura 11.13 A relação *cliente*.

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

Figura 11.14 Estrutura de arquivos com agrupamento de múltiplas tabelas.

Essa consulta calcula a junção das relações depositante e cliente. Assim, para cada tupla de depositante, o sistema precisa localizar as tuplas de *cliente* com o mesmo valor para *nome-cliente*. O ideal é que esses registros sejam localizados com a ajuda de *índices*, sobre os quais discutiremos no Capítulo 12. Contudo, independente de como esses registros são localizados, eles precisam ser transferidos do disco para a memória principal. No pior dos casos, cada registro residirá em um bloco diferente, forçando-nos a realizar uma leitura de bloco para cada registro solicitado pela consulta.

Como um exemplo concreto, considere as relações *depositante* e *cliente* das Figuras 11.12 e 11.13, respectivamente. Na Figura 11.14, mostramos uma estrutura de arquivos projetada para a execução eficiente de consultas envolvendo *depositante* ⊗ *cliente*. As tuplas de *depositante* para cada *nome-cliente* são armazenadas próximas da tupla de *cliente* para o *nome-cliente* correspondente. Essa estrutura mistura tuplas de duas relações, mas permite o processamento eficiente da junção. Quando uma tupla da relação *cliente* é lida, o bloco inteiro que contém essa tupla é copiado do disco para a memória principal. Como as tuplas de *depositante* correspondentes estão armazenadas no disco perto da tupla *cliente*, o bloco contendo a tupla *cliente* contém tuplas da relação *depositante* necessárias para processar a consulta. Se um cliente tiver tantas contas que os registros de *depositante* não caibam em um bloco, os registros restantes aparecerão em blocos nas proximidades.

Uma **organização de arquivos com agrupamento de múltiplas tabelas** é uma organização de arquivos, como aquela ilustrada na Figura 11.14, que armazena registros relacionados de duas ou mais relações em cada bloco. Essa organização de arquivos nos permite ler registros que satisfariam a condição de junção usando uma leitura de bloco. Assim, podemos processar essa consulta em particular com mais eficiência.

Nosso uso do agrupamento de múltiplas tabelas em um único arquivo melhorou o processamento de uma junção em particular (*depositante* ⊗ *cliente*), mas resulta em um processamento mais lento dos outros tipos de consulta. Por exemplo,

```
select *
from cliente
```

exige mais acessos de bloco do que acontecia no esquema sob o qual armazenamos cada relação em um arquivo separado. Em vez de vários registros *cliente* aparecendo em um bloco, cada registro é localizado em um bloco distinto. Na realidade, a simples localização de todos os registros *cliente* não é possível sem alguma estrutura adicional. Para localizar todas as tuplas da relação *cliente* na estrutura da Fi-

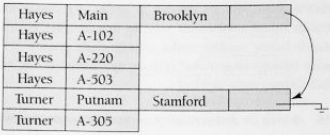


Figura 11.15 Estrutura de arquivos com agrupamento de múltiplas tabelas e cadeias de ponteiros.

gura 11.14, podemos encadear todos os registros dessa relação usando ponteiros, como na Figura 11.15.

Quando o agrupamento de múltiplas tabelas deve ser usado depende dos tipos de consulta que o projetista do banco de dados acredita que sejam mais freqüentes. O uso cuidadoso do agrupamento de múltiplas tabelas pode produzir ganhos de desempenho significativos no processamento da consulta.

Armazenamento em dicionário de dados

Até aqui, consideramos apenas a representação das próprias relações. Um sistema de banco de dados relacional precisa manter dados *sobre* as relações, como o esquema das relações. Essa informação é chamada **dicionário de dados** ou **catálogo do sistema**. Entre os tipos de informações que o sistema precisa armazenar estão estas:

- Nomes das relações
- Nomes dos atributos de cada relação
- Domínios e tamanhos dos atributos
- Nomes de visões definidas no banco de dados, e definições dessas visões
- As restrições de integridade (por exemplo, as restrições de chave)

Além disso, muitos sistemas mantêm os seguintes dados sobre usuários do sistema

- Nomes de usuários autorizados
- Informações de autorização e contabilidade sobre usuários
- Senhas ou outras informações usadas para autenticar usuários

O banco de dados também pode armazenar dados estatísticos e descritivos sobre as relações, como:

- Número de tuplas em cada relação
- Método de armazenamento para cada relação (por exemplo, agrupada ou não agrupada)

O dicionário de dados também pode anotar a organização do armazenamento (seqüencial, hash ou heap) das relações e o local onde cada relação está armazenada:

- Se as relações forem armazenadas nos arquivos do sistema operacional, o dicionário anota os nomes do arquivo (ou arquivos) contendo cada relação.
- Se o banco de dados armazena todas as relações em um único arquivo, o dicionário pode anotar os blocos contendo registros de cada relação em uma estrutura de dados, como uma lista interligada.

No Capítulo 12, em que estudamos índices, veremos uma necessidade de armazenar informações sobre cada índice em cada uma das relações:

- Nome do índice
- Nome da relação sendo indexada
- Atributos em que o índice é definido
- Tipo de índice formado

Toda essa informação constitui, com efeito, um banco de dados em miniatura. Alguns sistemas de banco de dados armazenam essas informações usando estruturas de dados de uso especial e código. Geralmente é preferível armazenar os dados sobre o banco de dados no próprio banco de dados. Usando o banco de dados para armazenar dados do sistema, simplificamos a estrutura geral do sistema e aumentamos o poder total do banco de dados para o acesso rápido aos dados do sistema.

A escolha exata de como representar os dados do sistema por relações precisa ser feita pelos projetistas do sistema. Uma representação possível, com as chaves primárias sublinhadas, aparece na Figura 11.16. Nessa representação, o atributo *atribs-índice* da relação *Metadados-índice* é considerada como tendo uma lista de um ou mais atributos, que podem ser representados por uma string de caracteres como "*nome-agência, cidade-agência*". A relação *Metadados-índice*, portanto, não está na primeira forma normal; ela pode ser normalizada, mas essa representação provavelmente será mais eficiente para acessar. O dicionário de da-

Metadados-relação (*nome_relação, número_de_atribs, org_armaz, local*)
Metadados-atributo (*nome_atrib, nome_relação, tipo_domínio, posição, tamanho*)
Metadados-usuário (*nome_usuário, senha_criptografada, grupo*)
Metadados-índice (*nome_índice, nome_relação, tipo_índice, atribs_índice*)
Metadados-visão (*nome_visão, definição*)

Figura 11.16 Banco de dados relacional representando dados do sistema.

dos normalmente é armazenado em uma forma não normalizada para alcançar acesso rápido.

A organização e o local do armazenamento dos próprios *Metadados-relação* precisam ser registrados em outro lugar (por exemplo, no próprio código do banco de dados), pois precisamos dessa informação para encontrar o conteúdo dos *Metadados-relação*.

Resumo

- Existem vários tipos de armazenamento de dados na maioria dos sistemas de computador. Eles são classificados pela velocidade com que podem acessar dados, por seu custo por unidade de dados para comprar a memória e por sua confiabilidade. Entre os meios disponíveis estão cache, memória principal, memória flash, discos magnéticos, discos ópticos e fitas magnéticas.
- Dois fatores determinam a confiabilidade do meio de armazenamento: se uma falha de energia ou falha do sistema causa perda de dados e qual é a probabilidade de falha física do dispositivo de armazenamento.
- Podemos reduzir a probabilidade de falha física retendo várias cópias dos dados. Para os discos, podemos usar o espelhamento. Ou então podemos usar métodos mais sofisticados com base em arrays redundantes de discos independentes (RAIDs). Espalhando dados pelos discos, esses métodos oferecem altas taxas de vazão em grandes acessos; introduzindo a redundância nos discos, eles melhoram muito a confiabilidade. Várias organizações RAID diferentes são possíveis, cada uma com diferentes características de custo, desempenho e confiabilidade. RAID nível 1 (espelhamento) e RAID nível 5 são os mais utilizados.
- Uma forma de reduzir a quantidade de acessos ao disco é manter o máximo de blocos possível na memória principal. Como não é possível manter todos os blocos na memória principal, precisamos gerenciar a alocação do espaço disponível na memória principal para o armazenamento de blocos. O *buffer* é aquela parte da memória principal disponível para o armazenamento de cópias de blocos de disco. O subsistema responsável pela alocação de espaço em buffer é chamado de *gerenciador de buffer*.
- Podemos organizar um arquivo logicamente como uma seqüência de registros mapeados nos blocos de disco. Uma técnica para mapear o banco de dados em arquivos é usar vários arquivos e armazenar registros de apenas um tamanho fixo em qualquer arquivo. Uma alternativa é estruturar os arquivos de modo que eles possam acomodar vários tamanhos para os registros. O método de página em slot é muito usado para lidar com registros de tamanho variável dentro de um bloco de disco.
- Como os dados são transferidos entre o armazenamento de disco e a memória principal em unidades de bloco, vale a pena atribuir registros de arquivo aos blocos de modo que um único bloco contenha registros relacionados. Se pudermos acessar vários dos registros que queremos com apenas um acesso de bloco, economizamos acessos ao disco. Como os acessos ao disco normalmente são o gargalo no desempenho de um sistema de banco de dados, a atribuição cuidadosa de registros aos blocos pode render importantes dividendos de desempenho.
- O dicionário de dados, também conhecido como catálogo do sistema, registra metadados, que são dados sobre dados, como nomes de relação, nomes e tipos de atributo, informação de armazenamento, restrições de integridade e informações de usuário.

Termos de revisão

- Meio de armazenamento físico
  - Cache
  - Memória principal
  - Memória flash
  - Disco magnético
  - Armazenamento óptico
- Disco magnético
  - Superfície
  - Discos rígidos
  - Disquetes
  - Trilhas
  - Setores
  - Cabeça de leitura-escrita
  - Braço de disco
  - Cilindro
  - Controladora de disco

- Somas de verificação (checksum)
- Remapeamento de setores defeituosos
- Medidas de desempenho dos discos
  - Tempo de acesso
  - Tempo de busca
  - Latência rotacional
  - Taxa de transferência de dados
  - Tempo médio para a falha (MTTF)
- Bloco de disco
- Otimização de acesso ao bloco de disco
  - Escalonamento do braço do disco
  - Algoritmo do elevador
  - Organização de arquivos
  - Desfragmentação
  - Buffers de escrita não voláteis
  - Memória de acesso aleatório não volátil (NVRAM)
  - Disco de log
- Array redundante de discos independentes (RAID)
  - Espelhamento
  - Espalhamento de dados
  - Espalhamento no nível de bit
  - Espalhamento no nível de bloco
- Níveis de RAID
  - Nível 0 (espalhamento de blocos, sem redundância)
  - Nível 1 (espalhamento de blocos, espelhamento)
  - Nível 3 (espalhamento de bits, paridade)
  - Nível 5 (espalhamento de blocos, paridade distribuída)
  - Nível 6 (espalhamento de blocos, redundância P + Q)
- Desempenho de reconstrução
- RAID de software
- RAID de hardware
- Troca a quente
- Armazenamento terciário
  - Discos ópticos
  - Fitas magnéticas
  - Jukeboxes
- Buffer
  - Gerenciador de blocos
  - Blocos presos
  - Saída forçada de blocos
- Políticas de substituição de buffer
  - Usado menos recentemente (LRU)
  - Lançar imediatamente
  - Usado mais recentemente (MRU)
- Arquivo
- Organização de arquivo
  - Cabeçalho de arquivo
  - Lista livre
- Registros de tamanho variável
  - Estrutura de página em slot
- Objetos grandes
- Organização de arquivos em heap
- Organização seqüencial de arquivos
- Organização de arquivos em hash
- organização por agrupamento de arquivos de múltiplas tabelas
- Chave de busca
- Dicionário de dados
- Catálogo do sistema

Exercícios práticos

11.1 Considere os seguintes dados e arrumação de bloco de paridade em quatro discos:

Disco 1	*Disco2*	*Disco 3*	*Disco 4*
B_1	B_2	B_3	B_4
P_1	B_5	B_6	B_7
B_8	P_2	B_9	B_{10}
⋮	⋮	⋮	⋮

Os *Bis* representam blocos de dados; os P_is representam os blocos de paridade. O bloco de paridade P_i é o bloco de paridade dos blocos B_{4i-3} até B_{4i}. Que problema (se houver algum) esse arranjo poderia apresentar?

11.2 Uma falta de energia que ocorre enquanto um bloco de disco está sendo gravado poderia resultar no bloco sendo escrito apenas parcialmente. Considere que os blocos parcialmente gravados podem ser detectados. Uma gravação de bloco indivisível é aquela em que ou o bloco de disco é totalmente gravado ou nada é gravado (ou seja, não existem gravações parciais). Sugira esquemas para conseguir o efeito de gravações de bloco indivisíveis com os seguintes esquemas RAID. Seus esquemas deverão envolver o trabalho com a recuperação de falhas.

a. RAID nível 1 (espelhamento)

b. RAID nível 5 (bloco intercalado, paridade distribuída)

11.3 Dê um exemplo de uma expressão da álgebra relacional e uma estratégia de processamento de consulta em cada uma das seguintes situações:

a. MRU é preferível a LRU.

b. LRU é preferível a MRU.

11.4 Considere a exclusão do registro 5 do arquivo da Figura 11.7. Compare os méritos relativos das seguintes técnicas para implementar a exclusão:

a. Mova o registro 6 para o espaço ocupado pelo registro 5, e mova o registro 7 para o espaço ocupado pelo registro 6.

b. Mova o registro 7 para o espaço ocupado pelo registro 5.

c. Marque o registro 5 conforme excluído, e não mova registros.

11.5 Mostre a estrutura do arquivo da Figura 11.8 após cada uma das seguintes etapas:
   a. Inserir(Brighton, A-323, 1600).
   b. Excluir registro 2.
      Delete record 2.
   c. Inserir(Brighton, A-626, 2000).

11.6 Considere o banco de dados relacional com duas relações:

curso (*nome-curso*, *sala*, *instrutor*)
matricula (*nome-curso*, *nome-aluno*, *série*)

Defina instâncias dessas relações para três cursos, cada um matriculando cinco alunos. Dê uma estrutura de arquivo dessas relações que utilize o agrupamento de múltiplas tabelas.

11.7 Considere a seguinte técnica de mapa de bits para acompanhar o espaço livre em um arquivo. Para cada bloco no arquivo, dois bits são mantidos no mapa de bits. Se o bloco estiver entre 0 e 30% cheio, os bits são 00; entre 30 e 60%, os bits são 01; entre 60 e 90%, os bits são 10; e acima de 90%, os bits são 11. Esses mapas de bit podem ser mantidos na memória até mesmo para arquivos muito grandes.
   a. Descreva como manter o mapa de bits atualizado nas inserções e exclusões de registro.
   b. Esboce o benefício da técnica de mapa de bits por listas livres na procura de espaço livre e na atualização da informação do espaço livre.

Exercícios

11.8 Liste os meios de armazenamento físico disponíveis nos computadores que você usa rotineiramente. Dê a velocidade com que os dados podem ser acessados em cada meio.

11.9 Como o remapeamento de setores defeituosos por controladores de disco afeta as taxas de recuperação de dados?

11.10 Os sistemas RAID normalmente permitem que você substitua discos defeituosos sem interromper o acesso ao sistema. Assim, os dados no disco defeituoso precisam ser reconstruídos e escritos no disco substituto enquanto o sistema está em operação. Qual dos níveis de RAID gera a menor quantidade de interferência entre a reconstrução e os acessos contínuos ao disco? Explique sua resposta.

11.11 Explique por que a alocação de registros aos blocos afeta tanto o desempenho do sistema de banco de dados.

11.12 Se possível, determine a estratégia de gerenciamento de buffer usada pelo sistema operacional em uso no seu sistema de computador local e que mecanismos ele oferece para controlar a substituição de páginas. Discuta como o controle de substituição que ele oferece seria útil para a implementação de sistemas de banco de dados.

11.13 Na organização de arquivo seqüencial, porque é usado um *bloco de estouro* se existir, no momento, apenas um registro de estouro?

11.14 Liste duas vantagens e duas desvantagens de cada uma das seguintes estratégias para armazenar um banco de dados relacional:
   a. Armazenar cada relação em um arquivo.
   b. Armazenar várias relações (talvez até mesmo o banco de dados inteiro) em um arquivo.

11.15 Dê uma versão normalizada da relação *Metadados-índice* e explique por que o uso da versão normalizada resultaria em um desempenho pior.

11.16 Se você tiver dados que não deverão ser perdidos em caso de falha do disco e estes são gravados com freqüência, como você os armazenaria?

11.17 Nos discos da geração anterior, a quantidade de setores por trilha era igual em todas as trilhas. A geração atual de discos tem mais setores por trilha nas bordas externas e menos setores nas trilhas interiores (pois possuem uma extensão menor). Qual é o efeito dessa mudança em cada um dos três principais indicadores de velocidade do disco?

11.18 Os gerenciadores de buffer padrão consideram que cada página tem o mesmo tamanho e custa o mesmo para ser lida. Considere um gerenciador de buffer que, em vez de LRU, utiliza a velocidade de referência aos objetos, ou seja, com que freqüência um objeto foi acessado nos últimos n segundos. Suponha que queremos armazenar no buffer objetos de tamanhos variáveis e custos de leitura variáveis (como páginas Web, cujo custo de leitura depende do site do qual elas são apanhadas). Sugira como um gerenciador de buffer pode escolher qual página deve ser expulsa do buffer.

Notas bibliográficas

Hennessy e *et al.* [2002] é um livro-texto popular sobre arquitetura de computador, que inclui a cobertura dos aspectos de hardware dos buffers de tradução look-aside, caches e unidades de gerenciamento de memória. Rosch [2003] apresenta uma visão geral excelente do hardware do computador, incluindo a cobertura extensiva de todos os tipos de tecnologia de armazenamento, como discos magnéticos, discos ópticos, fitas e interfaces de armazenamento. Patterson [2004] oferece uma boa discussão sobre como as melhorias na latência estão atrasadas em relação às melhorias na largura de banda (taxa de transferência).

Com o rápido aumento das velocidades de CPU, a memória cache localizada com a CPU tornou-se muito mais rápida do que a memória principal. Embora os sistemas de banco de dados não controlem quais dados são mantidos em cache, existe uma motivação crescente para organizar os dados na memória e escrever programas de modo que a utilização do cache seja maximizada. O trabalho nesse sentido inclui Rao e Ross [2000], Ailamaki *et al.* [2001] e Zhou e Ross [2004].

As especificações das unidades de disco da geração atual podem ser obtidas a partir de sites de seus fabricantes, como Hitachi, IBM (que recentemente vendeu sua fabricação de discos para a Hitachi), Seagate e Maxtor.

O espalhamento de disco foi descrito por Salem e Garcia-Molina [1986]. As discussões dos arrays redundantes de discos independentes (RAID) são apresentadas por Patterson *et al.* [1988] e por Chen e Patterson [1990]. Chen *et al.* [1994] apresentam um excelente levantamento dos princípios e da implementação do RAID. Códigos Reed-Solomon são abordados em Pless [1998]. Organizações de disco alternativas, que oferecem um alto grau de tolerância a falhas, incluem aquelas descritas por Gray *et al.* [1990] e Bitton e Gray [1988]. O sistema de arquivos baseado em log, que faz gravações seqüenciais em disco, é descrito em Rosenblum e Ousterhout [1991].

Em sistemas que dão suporte à computação móvel, os dados podem ser enviados repetidamente por banco de dados. O meio de broadcast pode ser visto como um nível da hierarquia de armazenamento – como um disco de broadcast com alta latência. Essas questões são discutidas em Acharya *et al.* [1995]. O caching e o gerenciamento de buffer para a computação móvel são discutidos em Barbará e Imielinski [1994].

A estrutura de armazenamento de sistemas de banco de dados específicos, como IBM DB2, Oracle ou Microsoft SQL Server, é documentada em seus respectivos manuais do sistema.

O gerenciamento de buffer é discutido na maioria dos textos sobre sistema operacional, incluindo em Silberschatz *et al.* [2001]. Chou e DeWitt [1985] apresentam algoritmos para gerenciamento de buffer em sistemas de banco de dados e descrevem uma avaliação de desempenho.

CAPÍTULO 12

Indexação e hashing

Muitas consultas referenciam apenas uma pequena proporção dos registros em um arquivo. Por exemplo, uma consulta como "Encontrar todas as contas na agência Perryridge" ou "Encontrar o saldo do número de conta A-101" referencia apenas uma fração dos registros de conta. Não é eficiente para o sistema ler cada registro e verificar o campo *nome_agência* para encontrar o nome "Perryridge" ou verificar o campo *número-conta* para encontrar o valor A-101. O ideal é que o sistema seja capaz de localizar esses registros diretamente. Para permitir essas formas de acesso, projetamos estruturas especiais que associamos aos arquivos.

Conceitos básicos

Um índice para um arquivo em um sistema de banco de dados funciona quase da mesma forma que o índice deste livro. Se quisermos descobrir algo sobre um assunto em particular (especificado por uma palavra ou uma frase) neste livro, podemos procurar o assunto no índice ao final do livro, encontrar a página em que ele ocorre e depois ler as páginas para encontrar as informações que estamos procurando. As palavras no índice estão em ordem, facilitando a localização daquela que estamos procurando. Além do mais, o índice é muito menor do que o livro, reduzindo ainda mais o esforço necessário para encontrar as palavras que estamos procurando.

Os índices no sistema de banco de dados desempenham o mesmo papel dos índices de livro nas bibliotecas. Por exemplo, para apanhar um registro de *conta* dado o número da conta, o sistema de banco de dados pesquisaria um índice para descobrir em que bloco do disco o registro correspondente reside, e depois apanharia o bloco do disco, para obter o registro de *conta*.

Manter uma lista classificada de números de conta não funcionaria bem em muitos sistemas grandes de banco de dados, com milhões de contas, pois o próprio índice seria muito grande; além disso, embora manter um índice classificado reduza o tempo de busca, encontrar uma conta ainda pode ser um tanto demorado. Em vez disso, técnicas de indexação mais sofisticadas podem ser utilizadas. Discutiremos várias dessas técnicas neste capítulo.

Existem dois tipos básicos de índices:

- **Índices ordenados.** Baseados em uma ordem classificada dos valores.
- **Índices de hash.** Baseados em uma distribuição uniforme de valores por um intervalo de buckets (baldes). O bucket ao qual um valor é atribuído é determinado por uma função, chamada *função de hash*.

Vamos considerar várias técnicas para a indexação ordenada e o hashing. Nenhuma técnica é a melhor. Em vez disso, cada técnica é mais adequada a determinadas aplicações de banco de dados. Cada técnica precisa ser avaliada com base nestes fatores:

- **Tipos de acesso:** os tipos de acesso que são aceitos com eficiência. Os tipos de acesso podem incluir a localização de registros com um valor de atributo especificado e a localização de registros cujos valores de atributos se encontrem em um intervalo especificado.
- **Tempo de acesso:** o tempo gasto para encontrar determinado item de dados, ou conjunto de itens, usando a técnica em questão.
- **Tempo de inserção:** o tempo gasto para inserir um novo item de dados. Esse valor inclui o tempo gasto

para encontrar o local atual para inserir o novo item de dados, além do tempo gasto para atualizar a estrutura de índice.

- **Tempo de exclusão:** o tempo gasto para excluir um item de dados. Esse valor inclui o tempo gasto para localizar o item a ser excluído, além do tempo gasto para atualizar a estrutura de índice.
- **Espaço adicional:** o espaço adicional ocupado por uma estrutura de índice. Desde que a quantidade de espaço adicional seja moderada, normalmente vale a pena sacrificar o espaço para conseguir um desempenho melhor.

Normalmente, queremos ter mais de um índice para um arquivo. Por exemplo, podemos querer pesquisar um livro por autor, por assunto ou por título.

Um atributo ou conjunto de atributos utilizados para pesquisar registros em um arquivo é denominado **chave de busca**. Observe que essa definição de *chave* difere daquela usada na *chave primária, chave candidata* e *superchave*. Esse duplo significado para *chave* (infelizmente) está bem estabelecido na prática. Usando nossa noção de chave de busca, vemos que, se existirem vários índices em um arquivo, várias chaves de busca serão usadas.

Índices ordenados

Para obter um acesso aleatório rápido aos registros em um arquivo, podemos usar uma estrutura de índice. Cada estrutura de índice está associada a uma determinada chave de busca. Assim como o índice de um livro ou um catálogo da biblioteca, um índice ordenado armazena os valores das chaves de busca em ordem classificada e associa a cada chave de busca os registros que a contêm.

Os próprios registros no arquivo indexado podem ser armazenados em alguma ordem classificada, assim como os livros em uma biblioteca são armazenados de acordo com algum atributo, como o número decimal de Dewey. Um arquivo pode ter vários índices, em diferentes chaves de busca. Se o arquivo contendo os registros for ordenado seqüencialmente, um **índice de agrupamento** é um índice cuja chave de busca também define a ordem seqüencial do arquivo. Os índices de agrupamento também são chamados **índices primários**; o termo índice primário parece indicar um índice sobre uma chave primária, mas esses índices na verdade podem ser montados com base em qualquer chave de busca. A chave de busca de um índice de agrupamento normalmente é a chave primária, embora isso não seja necessariamente dessa forma. Os índices cuja chave de busca especifica uma ordem diferente da ordem seqüencial do arquivo são chamados **índices não de agrupamento**, ou índices **secundários**. Os termos "agrupado" e "não agrupado" normalmente são usados no lugar de "agrupamento" e "não de agrupamento".

Nas seções "Índices densos e esparsos" a "Atualização do índice", consideramos que todos os arquivos são ordenados seqüencialmente em alguma chave de busca. Tais arquivos, com um índice agrupado sobre a chave de busca, são chamados **arquivos seqüenciais indexados**. Eles representam um dos esquemas de índice mais antigos utilizados nos sistemas de banco de dados. Eles são projetados para aplicações que exigem tanto o processamento seqüencial do arquivo inteiro quanto o acesso aleatório a registros individuais. Na seção "Índices secundários", explicamos os índices secundários.

A Figura 12.1 mostra um arquivo seqüencial de registros de *conta* apanhados do nosso exemplo bancário. No exemplo da Figura 12.1, os registros são armazenados na ordem da chave de busca, com *nome_agência* sendo usado como chave de busca.

Índices densos e esparsos

Um **registro de índice**, ou **entrada de índice**, consiste em um valor da chave de busca e ponteiros para um ou mais re-

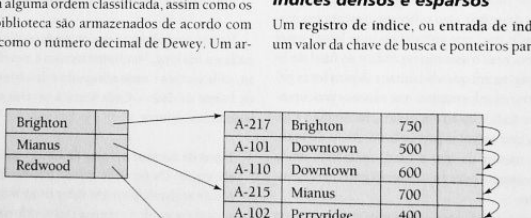


Figura 12.1 Arquivo seqüencial para registros de *conta*.

gistros com esse valor como seu valor de chave de busca. O ponteiro para um registro consiste no identificador de um bloco do disco e um deslocamento dentro do bloco de disco, para identificar o registro dentro do bloco.

Existem dois tipos de índices ordenados que podemos utilizar:

- **Índice denso:** um registro de índice aparece para cada valor de chave de busca no arquivo. Em um índice de agrupamento denso, o registro de índice contém o valor da chave de busca e um ponteiro para o primeiro registro de dados com esse valor da chave de busca. O restante dos registros com o mesmo valor da chave de busca seria armazenado seqüencialmente após o primeiro registro, pois, como o índice é agrupado, os registros são classificados sobre a mesma chave de busca.

  Implementações de índice denso podem armazenar uma lista de ponteiros para todos os registros com o mesmo valor de chave de busca; isso não é essencial para os índices agrupados.
- **Índice esparso:** um registro de índice aparece para somente alguns dos valores da chave de busca. Como acontece nos índices densos, cada registro de índice contém um valor de chave de busca e um ponteiro para o primeiro registro de dados com esse valor de chave de busca. Para localizar um registro, encontramos a entrada de índice com o maior valor de chave de busca que é menor ou igual ao valor de chave de busca pelo qual estamos procurando. Começamos no registro apontado pela entrada de índice e acompanhamos os ponteiros no arquivo até encontrarmos o registro desejado.

As Figuras 12.2 e 12.3 mostram os índices densos e esparsos, respectivamente, para o arquivo de *conta*. Suponha que estejamos procurando registros para a agência Perryridge. Usando o índice denso da Figura 12.2, acompanhamos o ponteiro diretamente para o primeiro registro de Perryridge. Processamos esse registro e acompanhamos o ponteiro nesse registro para localizar o próximo registro na ordem da chave de busca (*nome_agência*). Continuamos processando os registros até encontrarmos um registro para uma agência diferente de Perryridge. Se estivermos usando o índice esparso (Figura 12.3), não encontraremos uma entrada de índice para "Perryridge". Como a última entrada (em ordem alfabética) antes de "Perryridge" é "Mianus", acompanhamos esse ponteiro. Depois, lemos o arquivo *conta* em ordem seqüencial até encontrarmos o primeiro registro Perryridge, e começamos a processar nesse ponto.

Como vimos, geralmente é mais rápido localizar um registro se tivermos um índice denso ao invés de um índice esparso. Porém, os índices esparsos possuem vantagens em relação aos índices densos, pois exigem menos espaço e impõem menos sobrecarga de manutenção para inserções e exclusões.

Existe uma escolha que o projetista do sistema deve fazer entre tempo de acesso e espaço adicional. Embora a decisão com relação a isso dependa da aplicação específica, um bom meio-termo é usar um índice esparso com uma entrada de índice por bloco. O motivo para esse esquema ser uma boa escolha é que o custo dominante no processamento de uma solicitação de banco de dados é o tempo gasto para trazer um bloco do disco para a memória principal. Depois que o bloco for trazido, o tempo para varrer o bloco inteiro é mínimo. Usando esse índice esparso, localizamos o bloco que contém o registro que estamos buscando. Assim, a menos que o registro esteja em um bloco de estouro (ver seção "Organização seqüencial de arquivos" do Capítulo 11), reduzimos os acessos ao bloco enquanto mantemos o tamanho do índice (portanto, nossa sobrecarga de espaço) o menor possível.

Para que a técnica anterior seja totalmente genérica, temos de considerar o caso em que os registros para um valor de chave de busca ocupam vários blocos. É fácil modificar nosso esquema para lidar com essa situação.

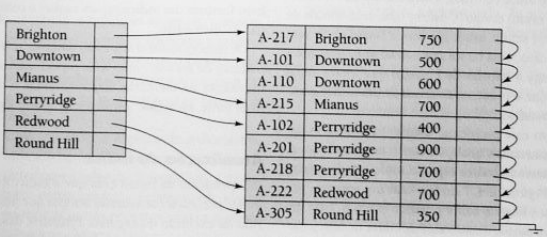


Figura 12.2 Índice denso.

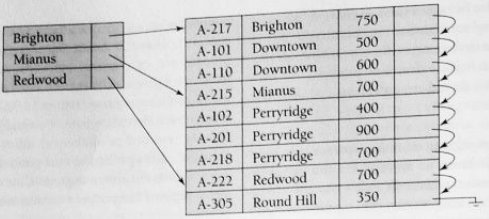


Figura 12.3 Índice esparso.

Índices multiníveis

Mesmo se usarmos um índice esparso, o próprio índice pode se tornar muito grande para um processamento eficiente. Na prática, não é razoável ter um arquivo com 100.000 registros, com 10 registros armazenados em cada bloco. Se tivermos um registro de índice por bloco, o índice terá 10.000 registros. Os registros de índice são menores do que os registros de dados, de modo que vamos considerar que 100 registros de índice cabem em um bloco. Assim, nosso índice ocupa 100 blocos. Esses grandes índices são armazenados como arquivos seqüenciais no disco.

Se um índice for suficientemente pequeno para ser mantido na memória principal, o tempo de busca para encontrar uma entrada será baixo. Porém, se o índice for tão grande que deva ser mantido no disco, uma busca por uma entrada exige várias leituras de bloco de disco. A busca binária pode ser usada sobre o arquivo de índice para localizar uma entrada, mas a busca ainda tem um grande custo. Se o índice ocupa b blocos, a busca binária exige até $\lceil \log_2(b) \rceil$ blocos a serem lidos. ($\lceil x \rceil$ indica o menor inteiro que seja maior ou igual a x; ou seja, arredondamos para cima.) Para o nosso índice de 100 blocos, a busca binária exige sete leituras de bloco. Em um sistema de disco em que uma leitura de bloco exige 30 milissegundos, a busca usará 210 milissegundos, o que é muito tempo. Observe que, se os blocos de estouro estiverem sendo usados, a busca binária não será possível. Nesse caso, uma busca seqüencial normalmente é usada, e isso exige b leituras de bloco, o que levará ainda mais tempo. Assim, o processo de busca de um índice grande pode ser dispendioso.

Para lidar com esse problema, tratamos o índice exatamente como trataríamos qualquer outro arquivo seqüencial, e construímos um índice esparso sobre o índice agrupado, como na Figura 12.4. Para localizar um registro, primeiro use a busca binária sobre o índice externo, para encontrar o registro para o maior valor de chave de busca menor ou igual a um que desejamos. O ponteiro aponta para um bloco do índice interno. Varremos esse bloco até encontrar o registro que tem o maior valor de chave de busca menor ou igual ao que desejamos. O ponteiro nesse registro aponta para o bloco do arquivo que contém o registro pelo qual estamos procurando.

Usando os dois níveis de indexação, lemos apenas um bloco de índice, em vez dos sete que lemos com a busca binária, se considerarmos que o índice externo já está na memória principal. Se o nosso arquivo for extremamente grande, até mesmo o índice externo pode crescer muito para caber na memória principal. Nesse caso, podemos criar outro nível de índice. Na verdade, podemos repetir esse processo tantas vezes quantas forem necessárias. Os índices com dois ou mais níveis são chamados de índices **multiníveis**. A busca de registros com um índice multinível exige muito menos operações de E/S do que a busca binária de registros. Cada nível de índice poderia corresponder a uma unidade de armazenamento físico. Assim, podemos ter índices nos níveis de trilha, cilindro e disco.

Um dicionário típico é um exemplo de um índice multinível em outro mundo, que não o de banco de dados. O cabeçalho de cada página lista a primeira palavra alfabeticamente nessa página. Esse tipo de índice de livro é um índice multinível: as palavras no alto de cada página do índice do livro formam um índice esparso sobre o conteúdo das páginas do dicionário.

Os índices multiníveis estão bastante relacionados às estruturas de árvore, como as árvores binárias usadas para a indexação na memória. Examinaremos o relacionamento mais tarde, na seção "Arquivos de índice de árvore B$^+$".

Atualização do índice

Independente da forma com que o índice é utilizado, cada índice precisa ser atualizado sempre que um registro é inserido ou excluído do arquivo. Primeiro, descrevemos os algoritmos para atualização de índices de único nível.

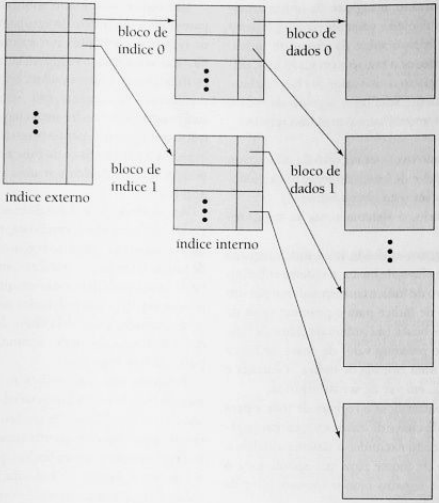


Figura 12.4 Índice esparso de dois níveis.

- Inserção. Primeiro, o sistema realiza uma pesquisa usando o valor de chave de busca que aparece no registro a ser inserido. As ações que o sistema realiza em seguida dependem se o índice é denso ou esparso:
  - □ Índices densos:
    1. Se o valor da chave de busca não aparece no índice, o sistema insere um registro de índice com o valor de chave de busca no índice, na posição apropriada.
    2. Caso contrário, a seguintes ações são tomadas:
       a. Se o registro de índice armazena ponteiros para todos os registros com o mesmo valor de chave de busca, o sistema acrescenta um ponteiro para o novo registro no registro de índice.
       b. Caso contrário, o registro de índice armazena um ponteiro somente no primeiro registro com o valor da chave de busca. O sistema, então, coloca o registro sendo inserido após os outros registros com os mesmos valores de chave de busca.
  - □ Índices esparsos: Consideramos que o índice armazena uma entrada para cada bloco. Se o sistema cria um novo bloco, ele insere no índice o primeiro valor de chave de busca (na ordem de chave de busca) que aparece no novo bloco. Por outro lado, se o novo registro tiver o menor valor de chave de busca em seu bloco, o sistema atualiza a entrada de índice apontando para o bloco; se não, o sistema não faz qualquer mudança no índice.
- Exclusão. Para excluir um registro, o sistema primeiro pesquisa o registro a ser excluído. As ações que o sistema toma em seguida dependem se o índice é denso ou esparso.
  - □ Índices densos:
    1. Se o registro excluído foi o único registro com seu valor específico de chave de busca, então o sistema retira do índice o registro de índice correspondente.
    2. Caso contrário, as seguintes ações são tomadas:
       a. Se o registro de índice armazena ponteiros para todos os registros com o mesmo valor de chave de índice, o sistema exclui do registro de índice o ponteiro para o registro excluído.

b. Caso contrário, o registro de índice armazena um ponteiro somente para o primeiro registro com o valor da chave de busca. Nesse caso, se o registro excluído foi o primeiro registro com o valor da chave de busca, o sistema atualiza o registro de índice para que aponte para o próximo registro.

□ Índices esparsos:

1. Se o índice não tiver um registro de índice com o valor de chave de busca do registro excluído, nada precisa ser feito com o índice.
2. Caso contrário, o sistema toma as seguintes ações:
   a. Se o registro excluído foi o único registro com sua chave de busca, o sistema substitui o registro de índice correspondente por um registro de índice para o próximo valor da chave de busca (na ordem de chave de busca). Se o próximo valor de chave de busca já tiver uma entrada de índice, a entrada é excluída, em vez de ser substituída.
   b. Caso contrário, se o registro de índice para o valor da chave de busca apontar para o registro sendo excluído, o sistema atualiza o registro de índice para que aponte para o próximo registro com o mesmo valor da chave de busca.

Os algoritmos de inserção e exclusão para índices multinível são uma extensão simples do esquema que acabamos de descrever. Na exclusão ou inserção, o sistema atualiza o índice de nível mais baixo conforme descrevemos. Com relação ao segundo nível, o índice de menor nível é simplesmente um arquivo contendo registros – assim, se houver qualquer mudança no índice de menor nível, o sistema atualiza o índice de segundo nível conforme descrevemos. A mesma técnica se aplica a outros níveis do índice, se houver algum.

Índices secundários

Os índices secundários precisam ser densos, com uma entrada de índice para cada valor de chave de busca, e um ponteiro para cada registro no arquivo. Um índice agrupado pode ser esparso, armazenando apenas alguns dos valores da chave de busca, pois sempre é possível encontrar registros com valores intermediários da chave de busca por um acesso seqüencial a uma parte do arquivo, como já descrevemos. Se um índice secundário armazenar apenas algumas das chaves de busca, os registros com valores intermediários de chave de busca podem estar em qualquer lugar no arquivo e, em geral, não podemos encontrá-los sem pesquisar o arquivo inteiro.

Um índice secundário sobre uma chave candidata se parece com um índice de agrupamento denso, exceto que os registros apontados por valores sucessivos no índice não são armazenados seqüencialmente. Porém, em geral, os índices secundários podem ter uma estrutura diferente dos índices de agrupamento. Se a chave de busca de um índice agrupado não for uma chave candidata, é suficiente que o índice aponte para o primeiro registro com um valor específico para a chave de busca, pois os outros registros podem ser apanhados por uma varredura seqüencial do arquivo.

Ao contrário, se a chave de busca de um índice secundário não for uma chave candidata, não será suficiente apontar apenas para o primeiro registro com cada valor de chave de busca. Os registros restantes com o mesmo valor de chave de busca poderiam estar em qualquer lugar no arquivo, pois os registros são ordenados pela chave de busca do índice agrupado, e não pela chave de busca do índice secundário. Portanto, um índice secundário precisa ter ponteiros para todos os registros.

Podemos usar um nível extra de indireção para implementar índices secundários sobre chaves de busca que não são chaves candidatas. Os ponteiros nesse índice secundário não apontam diretamente para o arquivo. Em vez disso, cada um aponta para um bucket que contém ponteiros para o arquivo. A Figura 12.5 mostra a estrutura de um índice secundário que usa um nível extra de indireção para o arquivo de *conta*, sobre a chave de busca *saldo*.

Uma varredura seqüencial na ordem do índice agrupado é eficiente porque os registros no arquivo são armazenados fisicamente na mesma ordem do índice. Porém, não podemos (exceto em casos especiais raros) armazenar um arquivo fisicamente ordenado pela chave de busca do índice agrupado e pela chave de busca de um índice secundário. Como a ordem da chave secundária e a ordem da chave física são diferentes, se tentarmos varrer o arquivo seqüencialmente na ordem da chave secundária, a leitura de cada registro provavelmente exigirá a leitura de um novo bloco do disco, o que é muito lento.

O procedimento descrito anteriormente para exclusão e inserção também pode ser aplicado a índices secundários; as ações tomadas são aquelas descritas para índices densos armazenando um ponteiro para cada registro no arquivo. Se um arquivo tiver vários índices, sempre que o arquivo for modificado, *cada* índice terá de ser atualizado.

Os índices secundários melhoram o desempenho das consultas que usam chaves diferentes da chave de busca do índice agrupado. Porém, eles impõem uma sobrecarga significativa na modificação do banco de dados. O projetista de um banco de dados decide quais índices secundários são desejados com base em uma estimativa da freqüência relativa das consultas e modificações.

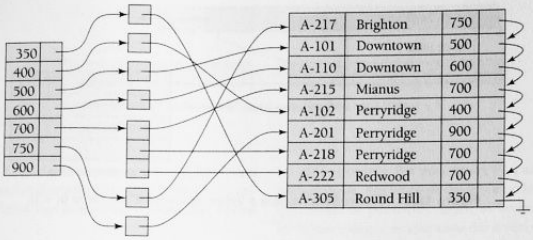


Figura 12.5 Índice secundário no arquivo de *conta*, sobre a chave não candidata *saldo*.

Arquivos de índice de árvore B⁺

A principal desvantagem da organização de arquivo seqüencial indexada é que o desempenho diminui enquanto o arquivo cresce, tanto para pesquisas de índice quanto para varreduras seqüenciais pelos dados. Embora essa degradação possa ser remediada pela reorganização do arquivo, as reorganizações freqüentes não são desejáveis.

A estrutura de índice de **árvore B⁺** é a mais utilizada das várias estruturas de índice que mantêm sua eficiência apesar da inserção e exclusão de dados. Um índice de árvore B⁺ tem a forma de uma **árvore balanceada**, em que cada caminho da raiz da árvore até uma folha da árvore é do mesmo tamanho. Cada nó não-folha na árvore tem entre $\lceil n/2 \rceil$ e n filhos, onde n é fixo para um árvore em particular.

Veremos que a estrutura de árvore B⁺ impõe sobrecarga de desempenho na inserção e na exclusão, além de aumentar o espaço adicional. A sobrecarga é aceitável até mesmo para arquivos modificados com freqüência, pois o custo da reorganização do arquivo é evitado. Além do mais, como os nós podem estar vazios até a metade (se tiverem o número mínimo de filhos), existe algum espaço desperdiçado. Esse espaço adicional também é aceitável dados os benefícios de desempenho da estrutura de árvore B⁺.

Estrutura de uma árvore B⁺

Um índice de árvore B⁺ é um índice de multinível, mas tem uma estrutura que difere daquela do arquivo seqüencial indexado multinível. A Figura 12.6 mostra um nó típico de uma árvore B⁺. Ele contém até $n - 1$ valores de chave de busca $K_1, K_2, \ldots, K_{n-1}$ e n ponteiros $P_1, P_2, \ldots, P_n$. Os valores de chave de busca dentro de um nó são mantidos em ordem classificada; assim, se $i < j$, então $K_i < K_j$.

Consideramos primeiro a estrutura dos nós de folha. Para $i = 1, 2, \ldots, n - 1$, o ponteiro P_i aponta para um registro de arquivo com valor de chave de busca K_i. A estrutura de bucket é usada somente se a chave de busca não formar uma chave candidata e se o arquivo não for classificado na ordem do valor da chave de busca. (Estudaremos mais tarde, na seção "Chaves de busca não exclusivas", como evitar a criação de buckets, fazendo com que a chave de busca pareça ser exclusiva.) O ponteiro P_n tem uma finalidade especial, que discutiremos em breve.

A Figura 12.7 mostra um nó de folha de uma árvore B⁺ para o arquivo de *conta*, em que escolhemos n como sendo 3, e a chave de busca é *nome_agência*. Observe que, como o arquivo de conta é ordenado por *nome_agência*, os ponteiros no nó de folha apontam diretamente para o arquivo.

Agora que vimos a estrutura de um nó de folha, vamos considerar como os valores da chave de busca são atribuídos aos nós em particular. Cada folha pode manter até $n - 1$ valores. Permitimos que os nós de folha contenham pelo menos $\lceil (n - 1)/2 \rceil$ valores. Os intervalos de valores em cada folha não se sobrepõem. Assim, se L_i e L_j são nós de folha e $i < j$, então cada valor de chave de busca em L_i é menor que cada valor de chave de busca em L_j. Se o índice de árvore B⁺ tiver de ser um índice denso, cada valor de chave de busca precisa aparecer em algum nó de folha.

Agora, podemos explicar o uso do ponteiro P_n. Como existe uma ordem linear nas folhas, com base nos valores de

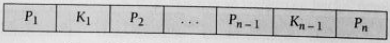


Figura 12.6 Nó típico de uma árvore B⁺.

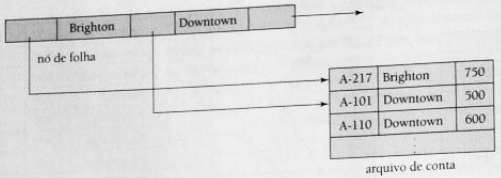


Figura 12.7 Um nó de folha para o índice de árvore B+ de *conta* ($n = 3$).

chave de busca que elas contêm, usamos P_n para encadear os nós de folha na ordem da chave de busca. Essa ordenação permite o processamento seqüencial eficiente do arquivo.

Os nós não-folha da árvore B+ formam um índice multinível (esparso) sobre os nós de folha. A estrutura dos nós não-folha é igual à dos nós de folha, exceto que todos os ponteiros são ponteiros para nós de folha. Um nó não-folha pode manter até n ponteiros, e *precisa* manter pelo menos $\lceil n/2 \rceil$ ponteiros. O número de ponteiros em um nó é chamado de *fanout* do nó.

Vamos considerar um nó contendo m ponteiros. Para $i = 2, 3, \ldots, m-1$, o ponteiro P_i aponta para a subárvore que contém valores de chave de busca menores que K_i e maiores ou iguais a K_{i-1}. O ponteiro P_m aponta para a parte da subárvore que contém os valores de chave maiores ou iguais a K_{m-1}, e o ponteiro P_1 aponta para a parte da subárvore que contém os valores de chave de busca menores que K_1.

Ao contrário dos outros nós de folha, o nó raiz pode manter menos de $\lceil n/2 \rceil$ ponteiros; porém, ele precisa manter pelo menos dois ponteiros, a menos que a árvore consista em apenas um nó. Sempre é possível construir uma árvore B+, para qualquer n, que satisfaça os requisitos anteriores. A Figura 12.8 mostra uma árvore B+ completa para o arquivo de *conta* ($n = 3$). Para simplificar, omitimos os dois ponteiros do próprio arquivo e os ponteiros nulos. Como exemplo de uma árvore B+ para a qual a raiz precisa ter menos de $\lceil n/2 \rceil$ valores, a Figura 12.9 mostra uma árvore B+ para o arquivo de *conta* com $n = 5$.

Esses exemplos de árvores B+ estão bem balanceados. Ou seja, o tamanho de cada caminho da raiz para o nó de folha é igual. Essa propriedade é um requisito para uma árvore B+. Na realidade, o "B" de árvore B+ significa "balanceada". É a propriedade de balanceamento das árvores B+ que garante o bom desempenho para pesquisa, inserção e exclusão.

Consultas em árvores B+

Vamos considerar como processar consultas sobre uma árvore B+. Suponha que queremos encontrar todos os registros com um valor de chave de busca V. A Figura 12.10 apresenta o pseudocódigo para fazer isso. Intuitivamente, o procedimento funciona da seguinte maneira. Primeiro, examinamos o nó raiz, procurando o menor valor de chave de busca que seja maior que V. Suponha que descobrimos que esse valor de chave de busca é K_i. Depois, acompanhamos o ponteiro P_i para outro nó. Se não encontrarmos esse valor, então $k \geq K_{m-1}$, onde m é o número de ponteiros no nó. Nesse caso, acompanhamos P_m para outro nó. No nó que alcançamos, novamente procuramos o menor valor de chave de busca maior que V, e mais uma vez seguimos o ponteiro correspondente, como antes. Por fim, alcançamos o nó de folha. No nó de folha, se encontrarmos o valor de

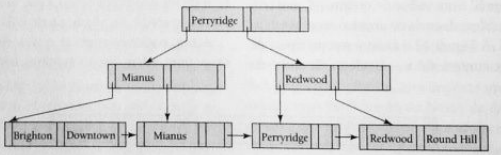


Figura 12.8 Árvore B+ para arquivo de *conta* (n = 3).

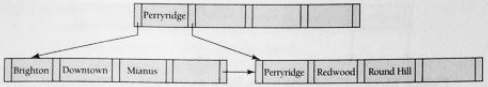


Figura 12.9 Árvore B^+ para arquivo de *conta* com $n = 5$.

chave de busca K_i igual a V, então o ponteiro P_i nos levará ao registro ou bucket desejado. Se o valor V não for encontrado no nó de folha, não existirá um registro com o valor de chave V.

Assim, no processamento de uma consulta, atravessamos um caminho na árvore desde a raiz até algum nó de folha. Se houver K valores de chave de busca no arquivo, o caminho não é maior do que $\lceil \log_{n/2}(K) \rceil$.

Na prática, somente alguns poucos nós precisarão ser acessados. Normalmente, um nó tem o mesmo tamanho de um bloco de disco, que normalmente é de 4 kilobytes. Com uma chave de busca com 12 bytes e um ponteiro de disco com 8 bytes, n fica em torno de 200. Mesmo com uma estimativa mais conservadora de 32 bytes para o tamanho da chave de busca, n fica em torno de 100. Com $n = 100$, se tivermos um milhão de valores de chave de busca no arquivo, uma pesquisa exigirá apenas $\lceil \log_{50}(1.000.000) \rceil = 4$ nós a serem acessados. Assim, no máximo quatro blocos precisam ser lidos do disco para a consulta. O nó raiz da árvore normalmente é muito acessado, e provavelmente estará no buffer, de modo que normalmente apenas três ou menos blocos precisam ser lidos do disco.

Uma diferença importante entre estruturas de árvore B^+ e estruturas de árvore na memória, como árvores binárias, é o tamanho de um nó e, como resultado, a altura da árvore. Em uma árvore binária, cada nó é pequeno e tem no máximo dois ponteiros. Em uma árvore B^+, cada nó é grande – normalmente, um bloco de disco – e pode ter uma grande quantidade de ponteiros. Assim, as árvores B^+ costumam ser baixas e largas, ao contrário das árvores binárias altas e estreitas. Em uma árvore binária balanceada, o caminho para uma pesquisa pode ter tamanho $\lceil \log_2(K) \rceil$, onde K é o número de valores de chave de busca. Com $K = 1.000.000$, como no exemplo anterior, uma árvore binária balanceada exige em torno de 20 acessos de nó. Se cada nó estivesse em um bloco de disco diferente, 20 leituras de bloco seriam necessárias para processar uma pesquisa, ao contrário das quatro leituras de bloco para a árvore B^+.

Atualizações em árvores B⁺

A inserções e a exclusão são mais complicadas do que a pesquisa, pois pode ser necessário **dividir** um nó que se torna muito grande como resultado de uma inserção, ou **unir** nós (ou seja, combinar nós) se um nó se tornar muito pequeno (menos de $\lceil n/2 \rceil$ ponteiros). Além do mais, quando um nó é dividido ou um par de nós é combinado, temos de garantir que o balanço seja preservado. Para introduzir a idéia por trás da inserção e exclusão em uma árvore B^+, vamos considerar temporariamente que os nós nunca se tornam muito grandes ou muito pequenos. Sob essa suposição, a inserção e a exclusão são realizadas conforme definido a seguir.

```
procedure find(valor V)
    set C = nó raiz
    while C não é o nó de folha begin
        Let K_i = menor valor de chave de busca > V, se houver
        if não existe um valor then begin
            Let m = número de ponteiros no nó
            set C = nó apontado por P_m
    end
            else set C = o nó apontado por P_i
    end
    if existe um valor de chave K_i em C tal que K_i = V
        then ponteiro P_i nos leva ao registro ou bucket desejado
        else não existe registro com valor de chave k
```

Figura 12.10 Consultando uma árvore B^+.

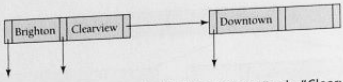


Figura 12.11 Divisão do nó de folha na inserção de "Clearview".

- **Inserção.** Usando a mesma técnica da pesquisa, encontramos o nó de folha em que o valor da chave de busca apareceria. Se o valor da chave de busca já aparecer no nó de folha, acrescentamos o novo registro ao arquivo e, se for preciso, acrescentamos ao bucket um ponteiro para o registro. Se o valor da chave de busca não aparecer, inserimos o valor no nó de folha e o posicionamos de modo que as chaves de busca ainda estejam na ordem. Depois, inserimos o novo registro no arquivo e, se for preciso, criamos um novo bucket com o ponteiro apropriado.
- **Exclusão.** Usando a mesma técnica da pesquisa, encontramos o registro a ser excluído e o removemos do arquivo. Removemos o valor da chave de busca do nó de folha se não houver um bucket associado a esse valor de chave de busca ou se o bucket ficar vazio como resultado da exclusão.

Agora, vamos considerar um exemplo em que um nó precisa ser dividido. Considere que queremos inserir um registro, com um valor de *nome_agência* igual a "Clearview", na árvore B+ da Figura 12.8. Usando o algoritmo para pesquisa, descobrimos que "Clearview" deverá aparecer no nó contendo "Brighton" e "Downtown". Portanto, o nó é *dividido* em dois nós. A Figura 12.11 mostra os dois nós de folha que resultam da inserção de "Clearview" e a divisão do nó contendo "Brighton" e "Downtown". Em geral, apanhamos os n valores de chave de busca (os $n - 1$ valores no nó de folha mais o valor sendo inserido) e colocamos os primeiros $\lceil n/2 \rceil$ no nó existente e os valores restantes em um novo nó.

Tendo dividido um nó de folha, temos de inserir o novo nó de folha na estrutura de árvore B+. Em nosso exemplo, o novo nó tem "Downtown" como seu menor valor de chave de busca. Temos de inserir esse valor de chave de busca no pai do nó de folha que foi dividido. A árvore B+ da Figura 12.12 mostra o resultado da inserção. O valor de chave de busca "Downtown" foi inserido no pai. Foi possível realizar essa inserção porque havia espaço para um valor de chave de busca adicional. Se não houvesse espaço, o pai teria de ser dividido. No pior caso, todos os nós ao longo do caminho até a raiz precisarão ser divididos. Se a própria raiz for dividida, a árvore inteira se torna mais profunda.

A técnica geral para inserção em uma árvore B+ é determinar o nó de folha f em que a inserção precisa ocorrer. Se houver uma divisão, insira o novo nó no pai do nó f. Se essa inserção causar uma divisão, suba recursivamente pela árvore até que uma inserção não cause uma divisão ou uma nova raiz seja criada.

A Figura 12.13 esboça o algoritmo de inserção em pseudocódigo. O procedimento insert insere um par de ponteiros de valor de chave no índice, usando dois procedimentos subsidiários, insert_in_leaf e insert_in_parent. No pseudocódigo, K, N, P e T indicam os ponteiros para os nós, com L sendo usado para indicar um nó de folha. $L.K_i$ e $L.P_i$ indicam o i-ésimo valor e o i-ésimo ponteiro no nó L, respectivamente; $T.K_i$ e $T.P_i$ são usados de modo semelhante. O pseudocódigo também utiliza a função *parent*(N) para encontrar o pai de um nó N. Podemos calcular uma lista de nós no caminho a partir da raiz até a folha enquanto encontramos inicialmente o nó de folha, e podemos usá-lo mais tarde para encontrar, de modo eficiente, o pai de qualquer nó no caminho.

O procedimento insert_in_parent recebe como parâmetros N, K', N', onde o nó N foi dividido em N e N', com K' sendo o valor de folha em N'. O procedimento modifica o pai de N para registrar a divisão. Os procedimentos insert_into_index e insert_in_parent utilizam uma área tem-

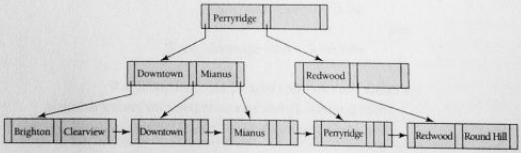


Figura 12.12 Inserção de "Clearview" na árvore B+ da Figura 12.8.

```
procedure insert(valor K, ponteiro P)
    encontre o nó de folha L que deve conter o valor de chave K
    if (L tem menos de n-1 valores de chave)
            then insert_in_leaf (L, K, P)
            else begin /* L tem n-1 valores de chave, divide */
                Crie novo L'
                Copie L.P1 ... L.Kn-1 para um bloco de memória T que
                        pode manter n pares (ponteiro, valor de chave)
                insert_in_leaf (T, K, P)
                Defina L'.Pn = L.Pn; Defina L.Pn = L'
                Apague L.P1 a L.Kn-1 de L
                Copie T.P1 a T.K⌈n/2⌉ de T para L começando em L.P1
                Copie T.P⌈n/2⌉+1 a T.Kn de T para L' começando em L'.P1
                Deixe K' ser o menor valor de chave em L'
                insert_in_parent(L, K', L')
            end

procedure insert_in_leaf (nó L, valor K, ponteiro P)
    if K é menor que L.K1
            then insira P, K em L logo antes de L.P1
            else begin
                Deixe Ki ser o menor valor em L que é menor que K
                insira P, K em L logo após T.Ki
            end

procedure insert_in_parent(nó N, valor K', nó N')
    if N é a raiz da árvore
            then begin
                crie novo nó R contendo N, K', N' /* N e N' são ponteiros */
                torne R a raiz da árvore
                return
            end
    Deixe P = parent(N)
    if (P tem menos de N ponteiros)
            then insira (K',N') em P logo após N
            else begin /* Divide P */
                Copie P para um bloco de memória T que pode manter P e (K',N')
                Insira (K', N') em T logo após N
                Apague todas as entradas de P; Crie nó P'
                Copie T.P1...T.P⌈n/2⌉ para P
                Deixe K'' = T.K⌈n/2⌉
                Copie T.P⌈n/2⌉+1...T.Pn+1 para P'
                insert_in_parent(P, K'', P')
            end
```

Figura 12.13 Inserção de entrada em uma árvore B+.

porária da memória T para armazenar o conteúdo de um nó sendo dividido. Os procedimentos podem ser modificados para copiar diretamente os dados do nó sendo dividido para o nó recém-criado, reduzindo o tempo necessário para copiar os dados. Porém, o uso do espaço temporário T simplifica os procedimentos.

Agora, consideramos as exclusões que fazem com que os nós de árvore contenham poucos ponteiros. Primeiro, va-

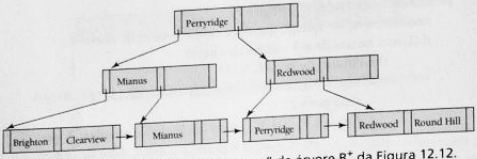


Figura 12.14 Exclusão de "Downtown" da árvore B+ da Figura 12.12.

mos excluir "Downtown" da árvore B+ da Figura 12.12. Localizamos a entrada para "Downtown" usando nosso algoritmo de pesquisa. Quando excluímos a entrada para "Downtown" do seu nó de folha, a folha fica vazia. Como, em nosso exemplo, $n = 3$ e $0 < \lceil (n-1)/2 \rceil$, esse código precisa ser eliminado da árvore B+. Para excluir um nó de folha, temos de excluir o ponteiro para ele a partir do seu pai. Em nosso exemplo, essa exclusão deixa o nó pai, que anteriormente tinha três ponteiros, com apenas dois ponteiros. Como $2 \geq \lceil n/2 \rceil$, o nó ainda é suficientemente grande, e a operação de exclusão está concluída. A árvore B+ resultante aparece na Figura 12.14.

Quando fazemos uma exclusão a partir de um pai de um nó de folha, o próprio nó pai pode se tornar muito pequeno. Isso é exatamente o que acontece se excluirmos "Perryridge" da árvore B+ da Figura 12.14. A exclusão da entrada Perryridge faz com que um nó de folha fique vazio. Quando excluímos o ponteiro para esse nó no pai deste, o pai fica com apenas um ponteiro. Como $n = 3, \lceil n/2 \rceil = 2$, e com isso apenas um ponteiro é muito pouco. Porém, como o nó pai contém informações úteis, não podemos simplesmente excluí-lo. Em vez disso, examinamos o nó irmão (o nó não-folha contendo a única chave de busca, Mianus). Esse nó irmão tem espaço para acomodar a informação contida em nosso nó agora pequeno, então unimos esses nós de modo que o nó irmão agora contenha as chaves "Mianus" e "Redwood". O outro nó (aquele contendo apenas a chave de busca "Redwood") agora contém informações redundantes e pode ser excluído do seu pai (que, em nosso exemplo, é a raiz). A Figura 12.15 mostra o resultado. Observe que a raiz tem apenas um ponteiro filho após a exclusão, de modo que é excluída e seu único filho se torna a raiz. Assim, a profundidade da árvore B+ foi diminuída em 1.

Nem sempre é possível unir nós. Como ilustração, exclua "Perryridge" da árvore B+ da Figura 12.12. Neste exemplo, a entrada "Downtown" ainda faz parte da árvore. Mais uma vez, o nó de folha contendo "Perryridge" torna-se vazio. O pai do nó de folha se torna muito pequeno (apenas um ponteiro). Porém, neste exemplo, o nó irmão já contém o número máximo de ponteiros: três. Assim, ele não pode acomodar um ponteiro adicional. A solução nesse caso é **redistribuir** os ponteiros de modo que cada irmão tenha dois ponteiros. O resultado aparece na Figura 12.16. Observe que a redistribuição de valores necessita de uma mudança de um valor de chave de busca no pai dos dois irmãos.

Em geral, para excluir um valor em uma árvore B+, realizamos uma pesquisa no valor e o excluímos. Se o nó for muito pequeno, nós o excluímos do seu pai. Essa exclusão resulta na aplicação recursiva do algoritmo de exclusão até que a raiz seja alcançada, um pai permanece adequadamente cheio após a exclusão, ou após a redistribuição ser aplicada.

A Figura 12.17 esboça o pseudocódigo para a exclusão a partir de uma árvore B+. O procedimento swap_variables(N,N') simplesmente troca os valores das variáveis (de ponteiro) N e N'; essa troca não tem efeito sobre a própria árvore. O pseudocódigo usa a condição "muito poucos ponteiros/valores". Para nós não de folha, esse critério significa menos de $\lceil n/2 \rceil$ ponteiros; para nós de folha, isso significa menos de $\lceil (n-1)/2 \rceil$ valores. O pseudocódigo redistribui as entradas pedindo emprestada uma única entrada de um nó adjacente. Também podemos redistribuir as entra-

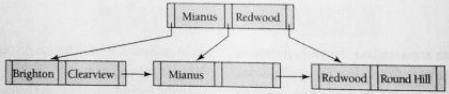


Figura 12.15 Exclusão de "Perryridge" da árvore B+ da Figura 12.14.

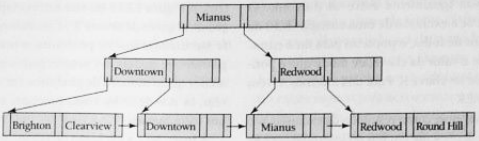


Figura 12.16 Exclusão de "Perryridge" da árvore B+ da Figura 12.12.

```
procedure delete(valor K, ponteiro P)
    encontre o nó de folha L que contém (K,P)
    delete_entry(L, K, P)

procedure delete_entry(nó N, valor K, ponteiro P)
    exclua (K,P) de N
    if (N é a raiz and N tem apenas um filho restante)
    then torna o filho de N a nova raiz da árvore e exclui N
    else if (N tem poucos valores/ponteiros) then begin
            deixe N' ser o filho anterior ou seguinte de parent(N)
            deixe K' ser o valor entre os ponteiros N e N' em parent(N)
            if (entradas em N e N' podem caber em um único nó)
                then begin /* Une nós */
                        if (N é predecessor de N') then swap_variables(N, N')
                        if (N não é uma folha)
                            then anexa K' e todos os ponteiros e valores em N a N'
                            else anexa todos os pares (K_i,P_i) em N a N'; define N'.P_n = N.P_n
                        delete_entry(parent(N), K', N); exclui nó N
                    end
            else begin /* Redistribuição: pega emprestada entrada de N' */
                if (N' é predecessor de N) then begin
                        if (N é um nó não de folha) then begin
                            deixe m ser tal que N'.P_m seja o último ponteiro em N'
                            remova (N'.K_m-1, N'.P_m) de N'
                            insira (N'.P_m, K') como primeiro ponteiro e valor em N,
                                    deslocando outros ponteiros e valores para a direita
                            substitua K' em parent(N) por N'.K_m-1
                        end
                        else begin
                            deixe m ser tal que (N'.P_m, N'.K_m) seja o último par
                                    ponteiro/valor em N'
                            remova (N'.P_m, N'.K_m) de N'
                            insira (N'.P_m, N'.K_m) como primeiro ponteiro e valor em N,
                                    deslocando outros ponteiros e valores para a direita
                            substitua K' em parent(N) por N'.K_m
                        end
                    end
                    else ... simétrico ao caso then ...
            end
    end
```

Figura 12.17 Exclusão da entrada de uma árvore B+.

das reparticionando-as igualmente entre os dois nós. O pseudocódigo refere-se à exclusão de uma entrada (K,P) de um nó. No caso de nós de folha, o ponteiro para uma entrada realmente precede o valor da chave, de modo que o ponteiro P precede o valor de chave K. Para nós internos, P vem após o valor de chave K.

Vale a pena observar que, como resultado da exclusão, um valor de chave que está presente em um nó interno da árvore B^+ pode não estar presente em qualquer folha da árvore.

Embora as operações de inserção e exclusão sobre as árvores B^+ sejam complicadas, elas exigem relativamente poucas operações de E/S, que é um benefício importante, pois as operações de E/S são dispendiosas. Pode-se mostrar que o número de operações de E/S necessárias para a inserção ou exclusão no pior caso é proporcional a $\log_{n/2}(K)$, onde n é o número máximo de ponteiros em um nó, e K é o número de valores da chave de busca. Em outras palavras, o custo das operações de inserção e exclusão é proporcional à altura da árvore B^+, e, portanto, é baixo. É a velocidade da operação sobre as árvores B^+ que as torna uma estrutura de índice freqüentemente utilizada nas implementações de banco de dados.

Organização de arquivos de árvore B^+

Conforme mencionamos na seção "Arquivos de índice de árvore B^+", a principal desvantagem da organização de arquivo seqüencial indexada é a degradação do desempenho quando o arquivo cresce: com o crescimento, uma porcentagem maior de registros de índice e registros reais se torna fora de ordem, e é armazenada em blocos de estouro. Solucionamos a degradação das pesquisas de índice usando índices de árvore B^+ no arquivo. Solucionamos o problema da degradação para armazenar os registros reais usando o nível de folha da árvore B^+ a fim de organizar os blocos que contêm os registros reais. Usamos a estrutura de árvore B^+ não apenas como um índice, mas também como um organizador para registros em um arquivo. Em uma **organização de arquivo de árvore B^+**, os nós de folha da árvore armazenam registros, em vez de armazenar ponteiros para registros. A Figura 12.18 mostra um exemplo de uma organização de arquivo de árvore B^+. Como os registros normalmente são maiores que os ponteiros, o número máximo de registros que podem ser armazenados em um nó de folha é menor que o número de ponteiros em um nó não-folha. Porém, os nós de folha ainda precisam estar cheios pelo menos até a metade.

A inserção e a exclusão de registros de uma organização de arquivo de árvore B^+ são tratadas da mesma maneira que a inserção e a exclusão de entradas em um índice de árvore B^+. Quando um registro com determinado valor de chave v é inserido, o sistema localiza o bloco que deverá conter o registro procurando na árvore B^+ a maior chave na árvore que é $\leq v$. Se o bloco localizado tiver espaço livre suficiente para o registro, o sistema armazena o registro no bloco. Caso contrário, como na inserção da árvore B^+, o sistema divide o bloco em dois e redistribui os registros nele (na ordem da chave da árvore B^+) para criar espaço para o novo registro. A divisão se propaga para cima na árvore B^+, pelo modo normal. Quando excluímos um registro, o sistema primeiro o remove do bloco que o contém. Se um bloco B ficar com menos de metade da ocupação como resultado, os registros em B são redistribuídos com os registros em um bloco B' adjacente. Considerando registros de tamanho fixo, cada bloco terá pelo menos metade da quantidade máxima de registros que pode admitir. O sistema atualiza os nós não de folha da árvore B^+ pelo modo normal.

Quando usamos uma árvore B^+ para a organização de arquivo, a utilização do espaço é particularmente importante, pois o espaço ocupado pelos registros provavelmente é muito mais do que o espaço ocupado pelas chaves e ponteiros. Podemos melhorar a utilização do espaço em uma árvore B^+ envolvendo mais nós irmãos na redistribuição durante as divisões e mesclagens. A técnica se aplica aos nós de folha e aos nós internos, e funciona da seguinte maneira.

Durante a inserção, se um nó estiver cheio, o sistema tenta redistribuir algumas de suas entradas para um dos nós adjacentes, a fim de criar espaço para uma nova entrada. Se essa tentativa falhar porque os próprios nós adjacentes estão cheios, o sistema divide o nó e divide as entradas

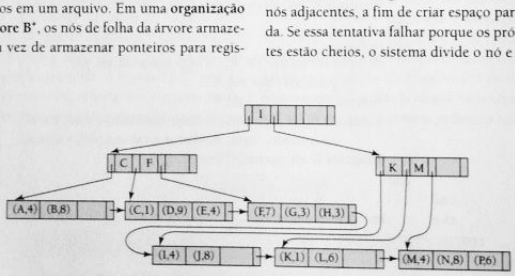


Figura 12.18 Organização de arquivo de árvore B^+.

por igual entre um dos nós adjacentes e os dois nós que obteve dividindo o nó original. Como os três nós juntos contêm mais um registro do que pode caber nos dois nós, cada nó estará cerca de dois terços cheio. Mais precisamente, cada nó terá pelo menos $\lfloor 2n/3 \rfloor$ entradas, onde n é o número máximo de entradas que o nó pode manter. ($\lfloor x \rfloor$ indica o maior inteiro que é menor ou igual a x; ou seja, removemos a parte fracionária, se houver.)

Durante a exclusão de um registro, se a ocupação de um nó ficar abaixo de $\lfloor 2n/3 \rfloor$, o sistema tenta pegar emprestada uma entrada de um dos nós irmãos. Se os dois nós irmãos tiverem $\lfloor 2n/3 \rfloor$ registros, em vez de pegar emprestada uma entrada, o sistema redistribui as entradas no nó e nos dois irmãos por igual entre dois dos nós, e exclui o terceiro nó. Podemos usar essa técnica porque o número total de entradas é $3\lfloor 2n/3 \rfloor$-1, que é menor que $2n$. Com três nós adjacentes sendo usados para a redistribuição, podemos garantir que cada nó terá $\lfloor 3n/4 \rfloor$ entradas. Em geral, se m nós (m-1 irmãos) estiverem envolvidos na redistribuição, pode-se garantir que cada nó terá pelo menos $\lfloor (m\text{-}1)n/m \rfloor$ entradas. Entretanto, o custo da atualização se torna maior à medida que mais nós irmãos são envolvidos na redistribuição.

Observe que, em um índice ou organização de arquivo de árvore B$^+$, os nós de folha que são adjacentes entre si na árvore podem estar localizados em diferentes locais do disco. Quando uma organização de arquivo é recém-criada em um conjunto de registros, é possível alocar blocos que são principalmente contíguos no disco aos nós de folha que são contíguos na árvore. Assim, uma varredura seqüencial dos nós de folha corresponderia a uma varredura geralmente seqüencial no disco. À medida que ocorrem inserções e exclusões na árvore, a seqüencialidade é perdida cada vez mais, e o acesso seqüencial precisa esperar pelas buscas de disco cada vez mais freqüentemente. Para restaurar a seqüencialidade, pode ser necessária uma reconstrução de índice.

Organizações de árvore B$^+$ podem ser usadas para armazenar grandes objetos, como clobs e blobs SQL, que podem ser maiores do que um bloco de disco, contendo até vários gigabytes. Esses objetos grandes podem ser armazenados dividindo-os em seqüências de registros menores que são colocados em uma organização de arquivo de árvore B$^+$. Os registros podem ser numerados seqüencialmente, ou numerados pelo deslocamento de bytes do registro dentro do objeto grande, e o número do registro pode ser usado como a chave de busca.

Indexando strings

A criação de índices de árvore B$^+$ sobre atributos com valor de string ocasiona dois problemas. O primeiro problema é que as strings podem ter tamanho variável. O segundo problema é que as strings podem ser longas, levando a um baixo fanout e uma altura de árvore conseqüentemente maior.

Com as chaves de busca de tamanho variável, diferentes nós podem ter diferentes fanouts, mesmo que estejam cheios. Um nó precisa ser dividido se estiver cheio, ou seja, não existe espaço para acrescentar uma nova entrada, independente de quantas entradas de busca ele tenha. De modo semelhante, os nós podem ser mesclados ou as entradas redistribuídas, dependendo da fração de espaço utilizada nos nós, em vez de nos basearmos no número máximo de entradas que o nó pode aceitar.

O fanout dos nós pode ser aumentado usando uma técnica chamada **compactação de prefixo**. Com a compactação de prefixo, não armazenamos o valor inteiro da chave de busca nos nós internos. Só armazenamos um prefixo de cada valor de chave de busca que seja suficiente para distinguir entre os valores de chave nas subárvores que ela separa. Por exemplo, se tivéssemos um índice sobre nomes, o valor de chave em um nó interno poderia ser um prefixo de um nome; pode ser suficiente armazenar "Silb" em um nó interno, em vez do nome "Silberschatz" completo, se os valores mais próximo nas duas subárvores que ela separa forem, digamos, "Silas" e "Silver", respectivamente.

Arquivos de índice de árvore B

Índices de árvore B são semelhantes aos índices de árvore B$^+$. A principal distinção entre as duas técnicas é que uma árvore B elimina o armazenamento redundante de valores de chave de busca. Na árvore B$^+$ da Figura 12.12, as chaves de busca "Downtown", "Mianus", "Redwood" e "Perryridge" aparecem duas vezes. Cada valor de chave de busca aparece em algum nó de folha; várias são repetidas nos nós não-folha.

Uma árvore B permite que os valores de chave de busca apareçam apenas uma vez. A Figura 12.19 mostra uma árvore B que representa as mesmas chaves de busca da árvore B$^+$ da Figura 12.12. Como as chaves de busca não são repetidas na árvore B, podemos armazenar o índice em menos nós de árvore do que no índice de árvore B$^+$ correspondente. Porém, como as chaves de busca que aparecem em nós não-folha não aparecem em outro lugar na árvore B, somos forçados a incluir um campo ponteiro adicional para cada chave de busca em um nó não-folha. Esses ponteiros adicionais apontam para os registros de arquivo ou buckets para a chave de busca associada.

Um nó de folha generalizado da árvore B aparece na Figura 12.20a; um nó não-folha aparece na Figura 12.20b. Os nós de folha são iguais aos das árvores B$^+$. Nos nós não-folha, os ponteiros P_i são os ponteiros de árvore que usamos também para árvores B$^+$, enquanto os ponteiros B_i são ponteiros de bucket ou registro de arquivo. Na árvore B generalizada da figura, existem n-1 chaves no nó de folha,

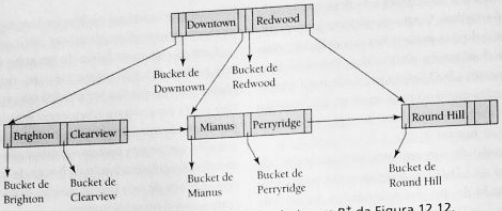


Figura 12.19 Árvore B equivalente da árvore B+ da Figura 12.12.

mas existem m-1 chaves no nó não-folha. Essa discrepância ocorre porque os nós não-folha precisam incluir ponteiros B_i, reduzindo assim o número de chaves de busca que podem ser mantidas nesses nós. Claramente, $m < n$, mas o relacionamento exato entre m e n depende do tamanho relativo das chaves de busca e ponteiros.

O número de nós acessados em uma pesquisa em uma árvore B depende de onde a chave de busca está localizada. Uma pesquisa sobre uma árvore B+ exige a travessia de um caminho da raiz da árvore até algum nó de folha. Ao contrário, às vezes é possível encontrar o valor desejado em uma árvore B antes de alcançar um nó de folha. Porém, aproximadamente n vezes mais chaves são armazenadas no nível de folha de uma árvore B do que nos níveis não-folha e, como n normalmente é grande, o benefício de localizar certos valores mais cedo é relativamente pequeno. Além do mais, o fato de que menos chaves de busca aparecem em um nó de árvore B não-folha, em comparação com as árvores B+, significa que uma árvore B tem um fanout menor e, portanto, pode ter profundidade maior do que a da árvore B+ correspondente. Assim, a pesquisa em uma árvore B é mais rápida para algumas chaves de busca, porém mais lenta para outras, embora, em geral, o tempo de pesquisa ainda seja proporcional ao logaritmo do número de chaves de busca.

A exclusão em uma árvore B é mais complicada. Em uma árvore B+, a entrada excluída sempre aparece em uma folha. Em uma árvore B, a entrada excluída pode aparecer em um nó não-folha. O valor correto precisa ser selecionado como um substituto a partir da subárvore do nó contendo a entrada excluída. Especificamente, se a chave de busca K_i for excluída, a menor chave de busca que aparece na subárvore do ponteiro P_{i+1} precisa ser movida para o campo anteriormente ocupado por K_i. Outras ações precisam ser tomadas se o nó de folha agora tiver muito poucas entradas. Ao contrário, a inserção em uma árvore B só é ligeiramente mais complicada do que a inserção em uma árvore B+.

As vantagens de espaço das árvores B são poucas para índices grandes, e normalmente não superam as desvantagens que observamos. Assim, muitos implementadores de sistema de banco de dados preferem a simplicidade estrutural de uma árvore B+. Os exercícios exploram os detalhes dos algoritmos de inserção e exclusão para árvores B.

Acesso por chave múltipla

Até agora, consideramos implicitamente que apenas um índice no atributo é usado para processar uma consulta em uma relação. Porém, para certos tipos de consultas, é vantajoso usar vários índices se eles existem, ou usar um índice baseado em uma chave de busca de múltiplos atributos.

Usando múltiplos índices de chave única

Suponha que o arquivo de *conta* tenha dois índices: um para *nome_agência* e um para saldo. Considere a seguinte

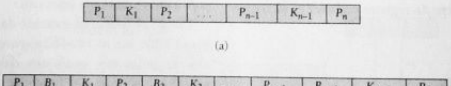


Figura 12.20 Nós típicos de uma árvore B. (a) Nó de folha. (b) Nó não de folha.

consulta: "Encontrar todos os números de conta na agência Perryridge com saldos iguais a $1.000". Escrevemos

```
select número_empréstimo
from conta
where nome_agência = "Perryridge" and saldo = 1000
```

Existem três estratégias possíveis para processar essa consulta:

1. Usar o índice sobre *nome_agência* e encontrar todos os registros pertencentes à agência Perryridge. Examinar cada um desses registros para ver se *saldo* = 1000.
2. Usar o índice sobre *saldo* para encontrar todos os registros pertencentes às contas com saldos iguais a $1.000. Examinar cada um desses registros para ver se *nome_agência* = "Perryridge".
3. Usar o índice sobre *nome_agência* para encontrar *ponteiros* para todos os registros pertencentes à agência Perryridge. Além disso, usar o índice sobre *saldo* para encontrar ponteiros para todos os registros pertencentes às contas com um saldo igual a $1.000. Apanhar a interseção desses dois conjuntos de ponteiros. Aqueles ponteiros que estão na interseção apontam para os registros pertencentes a Perryridge e contas com um saldo de $1.000.

A terceira estratégia é a única das três que tira proveito da existência de vários índices. Porém, até mesmo essa estratégia pode ser uma escolha fraca se acontecer o seguinte:

- Existem muitos registros pertencentes à agência Perryridge.
- Existem muitos registros pertencentes a contas com um saldo de $1.000.
- Existem apenas alguns registros pertencentes *tanto* à agência Perryridge quanto a contas com um saldo de $1.000.

Se essas condições forem mantidas, temos de varrer uma grande quantidade de ponteiros para produzir um resultado pequeno. Uma estrutura de índice chamada "índice de mapa de bits" pode, em alguns casos, agilizar bastante a operação de interseção usada na terceira estratégia. Os índices de mapa de bits são esboçados na seção "Índices de mapa de bits".

Índices sobre chaves múltiplas

Uma estratégia alternativa para esse caso é criar e usar um índice sobre uma chave de busca (*nome_agência, saldo*) – ou seja, a chave de busca consistindo no nome da agência concatenado com o saldo da conta. Tal chave de busca, contendo mais de um atributo, às vezes é chamada de **chave de busca composta**. A estrutura do índice é igual à de qualquer outro índice, mas a única diferença é que a chave de busca não é um único atributo, mas sim uma lista de atributos. A chave de busca pode ser representada como uma tupla de valores, da forma $(a_1, \ldots, a_n)$, onde os atributos indexados são $A_1, \ldots, A_n$. A ordenação dos valores da chave de busca é a ordenação *lexicográfica*. Por exemplo, para o caso de chaves de busca com dois atributos, $(a_1, a_2) < (b_1, b_2)$ se $a_1 < b_1$ ou $a_1 = b_1$ e $a_2 < b_2$. A ordenação lexicográfica é basicamente o mesmo que ordenação alfabética de palavras.

Podemos usar um índice ordenado (árvore B$^+$) para responder de forma eficiente a consultas da forma

```
select número_empréstimo
from conta
where nome_agência = 'Perryridge'
and saldo = 1000
```

Consultas como a seguinte, que especifica uma condição de igualdade sobre o primeiro atributo da chave de busca (*nome_agência*) e um intervalo sobre o segundo atributo da chave de busca (*saldo*), também podem ser tratadas de forma eficiente, pois correspondem a uma consulta de intervalo sobre o atributo de busca.

```
select número_empréstimo
from conta
where nome_agência = 'Perryridge'
and saldo < 1000
```

Podemos até mesmo usar um índice ordenado sobre a chave de busca (*nome_agência, saldo*) para responder à seguinte consulta sobre apenas um atributo de forma eficiente:

```
select número_empréstimo
from conta
where nome_agência = 'Perryridge'
```

Uma condição de igualdade *nome_agência* = "Perryridge" é equivalente a uma consulta de intervalo sobre o intervalo com a extremidade inferior (Perryridge, $-\infty$) e a extremidade superior (Perryridge, $+\infty$). As consultas com intervalo sobre apenas o atributo *nome_agência* podem ser tratadas de maneira semelhante.

Contudo, o uso de uma estrutura de índice ordenado sobre uma chave de busca composta tem algumas limitações. Como ilustração, considere a consulta

```
select número_empréstimo
from conta
where nome_agência < "Perryridge"
and saldo = 1000
```

Podemos responder a essa pergunta usando um índice ordenado sobre a chave de busca (*nome_agência, balance*): para cada valor de *nome_agência* que seja menor que "Perryridge" em ordem alfabética, o sistema localiza registros com um valor de *saldo* igual a 1.000. Porém, cada registro provavelmente estará em um bloco de disco diferente, devido à ordenação dos registros no arquivo, levando a muitas operações de E/S.

A diferença entre essa consulta e as duas anteriores é que a condição no primeiro atributo (*nome_agência*) é uma condição de comparação, em vez de uma condição de igualdade. A condição não corresponde a uma consulta de intervalo sobre a chave de busca.

Para agilizar o processamento de consultas gerais de chave de busca composta (que podem envolver uma ou mais operações de comparação), podemos usar várias estruturas especiais. Vamos considerar os *índices de mapa de bits* na seção "Índices de mapa de bits". Existe outra estrutura, chamada *árvore R*, que pode ser usada para essa finalidade. A árvore R é uma extensão da árvore B^+ para lidar com a indexação sobre várias dimensões. Como a árvore R é usada principalmente com tipos de dados geográficos, descrevemos a estrutura no Capítulo 24.

Chaves de busca não exclusivas

A criação de buckets de ponteiros de registro para lidar com chaves de busca nãoexclusivas (ou seja, chaves de busca que podem lidar com mais de um registro combinando) cria várias complicações quando as árvores B^+ são implementadas. Se os buckets forem mantidos no nó de folha, um código extra é necessário para lidar com os buckets de tamanho variável e para lidar com buckets que se tornam maiores do que o tamanho do nó de folha. Se os buckets forem armazenados em páginas separadas, uma operação de E/S extra pode ser necessária para apanhar os registros.

Uma solução simples para esse problema, usada pela maioria dos sistemas de banco de dados, é tornar as chaves de busca exclusivas, acrescentando um atributo exclusivo extra à chave de busca. O valor do atributo extra poderia ser uma id de registro (se o sistema de banco de dados admitir ids de registro), ou apenas um número que seja exclusivo entre todos os registros com o mesmo valor de chave de busca. Por exemplo, se tivéssemos um índice sobre o atributo *nome_cliente* da tabela *depositante*, as entradas correspondentes a um nome de cliente em particular teriam diferentes valores para o atributo extra. Com isso, a chave de busca estendida terá garantias de exclusividade.

Uma busca com o atributo de chave de busca original se torna uma pesquisa de intervalo sobre a chave de busca estendida, como vimos na seção anterior; o valor do atributo extra é ignorado durante a pesquisa.

Índices de cobertura

Índices de cobertura são índices que armazenam os valores de alguns atributos (fora os atributos da chave de busca) junto com os ponteiros do registro. Armazenar valores de atributo extras é útil com os índices secundários, pois nos permitem responder a algumas perguntas usando apenas o índice, sem sequer pesquisar os registros reais.

Por exemplo, suponha que tenhamos um índice não agrupado sobre o atributo *número_conta* da relação *conta*. Se armazenarmos o valor do atributo *saldo* junto com o ponteiro do registro, podemos responder a consultas que exijam o saldo (mas não o outro atributo, *nome_agência*) sem acessar o registro *conta*.

O mesmo efeito poderia ser obtido criando-se um índice sobre a chave de busca (*número_conta,saldo*), mas um índice de cobertura reduz o tamanho da chave de busca, permitindo um fanout maior nos nós internos, e potencialmente reduzindo a altura do índice.

Índices secundários e relocação de registros

Algumas organizações de arquivo, como a organização de arquivo de árvore B^+, podem mudar o local dos registros mesmo quando os registros não foram atualizados. Como um exemplo, quando uma página de folha é dividida em uma organização de arquivo de árvore B^+, uma série de registros são movidos para uma nova página. Nesses casos, todos os índices secundários que armazenam ponteiros para os registros relocados teriam de ser atualizados, embora os valores nos registros possam não ter alterado. Cada página de folha pode conter uma quantidade muito grande de registros, e cada um deles pode ser em locais diferentes em cada índice secundário. Assim, a divisão de página de folha pode exigir dezenas ou mesmo centenas de operações de E/S para atualizar todos os índices secundários afetados, tornando-a uma operação muito dispendiosa.

Uma técnica para lidar com esse problema é a seguinte. Nos índices secundários, no lugar de ponteiros para os registros indexados, armazenamos os valores dos atributos de chave de busca do índice primário. Por exemplo, suponha que tenhamos um índice primário sobre o atributo *número_conta* da relação *conta*; então, um índice secundário sobre *nome_agência* armazenaria com cada nome de agência uma

lista de valores de *número_conta* dos registros correspondentes, em vez de armazenar ponteiros para os registros.

A relocação de registros devido a divisões de página de folha não exige qualquer atualização sobre tal índice secundário. Porém, localizar um registro usando o índice secundário agora exige duas etapas: primeiro, usamos o índice secundário para encontrar os valores de chave de busca do índice primário, e depois usamos o índice primário para encontrar os registros correspondentes.

Essa técnica, portanto, reduz bastante o custo da atualização de índice, devido à reorganização do arquivo, embora aumente o custo do acesso aos dados, usando um índice secundário.

Hashing estático

Uma desvantagem da organização de arquivo seqüencial é que temos de acessar uma estrutura de índice para localizar dados, ou temos de usar a busca binária, e isso resulta em mais operações de E/S. As organizações de arquivo baseadas na técnica de **hashing** permitem evitar o acesso a uma estrutura de índice. O hashing também oferece um modo de construir índices. Estudaremos as organizações de arquivo e os índices baseados em hashing nas próximas seções.

Em nossa descrição do hashing, usaremos o termo **bucket** para indicar uma unidade de armazenamento que pode armazenar um ou mais registros. Um bucket normalmente é um bloco de disco, mas poderia ser menor ou maior do que um bloco de disco.

Formalmente, considere que K indica o conjunto de todos os valores de chave de busca, e que B indica o conjunto de todos os endereços de bucket. Uma **função de hash** h é uma função de K para B. Considere que h indica uma função de hash.

Para inserir um registro com chave de busca K_i, calculamos $h(K_i)$, que oferece o endereço do bucket para esse registro. Suponha, por enquanto, que existe espaço no bucket para armazenar o registro. Então, o registro é armazenado nesse bucket.

Para realizar uma pesquisa sobre o valor de chave de busca K_i, simplesmente calculamos $h(K_i)$, depois pesquisamos o bucket com esse endereço. Suponha que duas chaves de busca, K_5 e K_7, tenham o mesmo valor de hash; ou seja, $h(K_5) = h(K_7)$. Se realizarmos uma pesquisa sobre K_5, o bucket $h(K_5)$ contém registros com valores de chave de busca K_5 e registros com valores de chave de busca K_7. Assim, temos de verificar o valor de chave de busca de cada registro no bucket para verificar se o registro é aquele que desejamos.

A exclusão é igualmente simples. Se o valor de chave de busca do registro a ser excluído for K_i, calculamos $h(K_i)$, depois consultamos o bucket correspondente para esse registro e excluímos o registro do bucket.

O hashing pode ser usado para duas finalidades diferentes. Em uma **organização de arquivo de hash**, obtemos o endereço do bloco de disco contendo um registro desejado diretamente calculando uma função sobre o valor de chave de busca do registro. Em uma **organização de índice de hash**, organizamos as chaves de busca, com seus ponteiros associados, para uma estrutura de arquivo de hash.

Funções de hash

A pior função de hash possível mapeia todos os valores de chave de busca para o mesmo bucket. Essa função é indesejável porque todos os registros precisam ser mantidos no mesmo bucket. Uma pesquisa precisa examinar cada registro desse tipo para encontrar o desejado. A função de hash ideal distribui as chaves armazenadas uniformemente por todos os buckets, de modo que cada um tenha o mesmo número de registros.

Como não sabemos, durante o projeto, exatamente quais valores de chave de busca serão armazenados no arquivo, queremos escolher uma função de hash que atribua valores de chave de busca aos buckets de modo que a distribuição tenha estas qualidades:

- A distribuição é *uniforme*. Ou seja, a função de hash atribui a cada bucket o mesmo número de valores de chave de busca do conjunto de *todos* os valores de chave de busca possíveis.
- A distribuição é *aleatória*. Ou seja, no caso mais comum, cada bucket terá quase o mesmo número de valores atribuídos a ele, independente da distribuição real dos valores de chave de busca. Mais precisamente, o valor de hash não estará relacionado a qualquer ordenação externamente visível sobre os valores da chave de busca, como a ordenação alfabética ou a ordenação pelo tamanho das chaves de busca; a função de hash parecerá ser aleatória.

Como ilustração desses princípios, vamos escolher uma função de hash para o arquivo *conta* usando a chave de busca *nome_agência*. A função de hash que escolhemos precisa ter as propriedades desejáveis não apenas sobre o arquivo de exemplo *conta* que usamos, mas também sobre um arquivo *conta* do tamanho realista para um grande banco com muitas agências.

Suponha que decidimos ter 26 buckets e definimos uma função de hash que mapeia os nomes começando com a i-ésima letra do alfabeto para o i-ésimo bucket. A função de hash tem a virtude da simplicidade, mas não consegue oferecer uma distribuição uniforme, pois esperamos que mais nomes de agência comecem com as letras B e R, em vez de Q e X, por exemplo.

Agora suponha que queiramos uma função de hash sobre a chave de busca *saldo*. Suponha que o saldo mínimo seja 1 e o saldo máximo seja 100.000 e usemos uma função de hash que divida os valores em 10 intervalos, 1-10.000, 10.001-20.000 e assim por diante. A distribuição dos valores de chave de busca é uniforme (pois cada bucket tem o mesmo número de diferentes valores de *saldo*), mas não é aleatória. Contudo, os registros com saldos entre 1 e 10.000 são muito mais comuns do que os registros com saldos entre 90.001 e 100.000. Como resultado, a distribuição de registros não é uniforme – alguns buckets recebem mais registros do que outros. Se a função tiver uma distribuição aleatória, mesmo que tais correlações nas chaves de busca, a aleatoriedade da distribuição tornará muito provável que todos os buckets tenham aproximadamente o mesmo número de registros, desde que cada chave de busca ocorra apenas em uma pequena fração dos registros. (Se uma única chave de busca ocorrer em uma grande fração dos registros, o bucket que a contém provavelmente terá mais registros do que outros buckets, independente da função de hash utilizada.)

As funções de hash típicas realizam cálculos sobre a representação de máquina binária interna dos caracteres na chave de busca. Uma função de hash simples desse tipo primeiro calcula a soma das representações binárias dos caracteres de uma chave, depois retorna o módulo da soma dos números dos buckets. A Figura 12.21 mostra a aplicação desse tipo de esquema, com 10 buckets, ao arquivo *conta*, sob a suposição de que a *i*-ésima letra do alfabeto seja representada pelo inteiro *i*.

A seguinte função de hash pode ser usada para desmembrar uma string em uma implementação Java:

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \ldots + s[n-1]$$

A função pode ser implementada de forma eficiente definindo o resultado do hash inicialmente como 0, e percorrendo do primeiro ao último caractere da string, em cada etapa multiplicando o valor de hash por 31 e depois somando o caractere seguinte (tratado como um inteiro). O resultado do módulo dessa função pelo número de buckets pode ser usado para a indexação.

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Figura 12.21 Organização de hash do arquivo *conta*, com a chave *nome_agência*.

As funções de hash exigem um projeto cuidadoso. Uma função de hash errada pode resultar em um tempo de pesquisa proporcional ao número de chaves de busca no arquivo. Uma função bem projetada oferece um tempo de pesquisa médio constante (e pequeno), independente do número de chaves de busca no arquivo.

Tratamento de estouros de bucket

Até aqui, consideramos que, quando um registro é inserido, o bucket ao qual ele é mapeado possui espaço para armazenar o registro. Se o bucket não tiver espaço suficiente, acontece um **estouro de bucket**. O estouro de bucket pode ocorrer por vários motivos:

- **Buckets insuficientes**. O número de buckets, que indicamos com n_B, precisa ser escolhido de modo que $n_B > n_r/f_r$, onde n_r indica o número total de registros que serão armazenados e f_r indica o número de registros que caberão em um bucket. Essa designação, naturalmente, considera que o número total de registros é conhecido quando a função de hash é escolhida.
- **Distorção**. Alguns buckets recebem mais registros do que outros, de modo que um bucket pode estourar mesmo quando outros buckets ainda têm espaço. Essa situação é chamada de **distorção** de bucket. A distorção pode ocorrer por dois motivos:
  1. Vários registros podem ter a mesma chave de busca.
  2. A função de hash escolhida pode resultar na distribuição não uniforme das chaves de busca.

Para que a probabilidade de estouro de bucket seja reduzida, o número de buckets é escolhido como $(n_r/f_r)*(1 + d)$, onde d é um fator de fudge, normalmente em torno de 0,2. Algum espaço é desperdiçado: cerca de 20% do espaço nos baldes estará vazio. No entanto, o benefício é que a probabilidade de estouro é reduzida.

Apesar da alocação de mais alguns buckets do que o necessário, o estouro de bucket ainda pode ocorrer. Tratamos o estouro de bucket usando **buckets de estouro**. Se um registro tiver de ser inserido no bucket b, e b já estiver cheio, o sistema oferece um bucket de estouro para b e insere o registro no bucket de estouro. Se o bucket de estouro também estiver cheio, o sistema oferece outro bucket de estouro, e assim por diante. Todos os buckets de estouro de determinado bucket são encadeados em uma lista interligada, como na Figura 12.22. O tratamento do estouro usando tal lista interligada é chamado de **encadeamento de estouro**.

Temos de mudar o algoritmo de pesquisa ligeiramente para lidar com o encadeamento de estouro. Como antes, o sistema usa a função de hash sobre a chave de busca para identificar um bucket b. O sistema precisa examinar todos os registros no bucket b para ver se eles combinam com a chave de busca, como antes. Além disso, se o bucket b tiver buckets de estouro, o sistema também terá de examinar os registros em todos os buckets de estouro.

A forma da estrutura de hash que acabamos de descrever às vezes é chamada de **hashing fechado**. Sob uma técnica alternativa, chamada **hashing aberto**, o conjunto de buckets é fixo, e não existem cadeias de estouro. Em vez disso, se um bucket estiver cheio, o sistema insere registros em al-

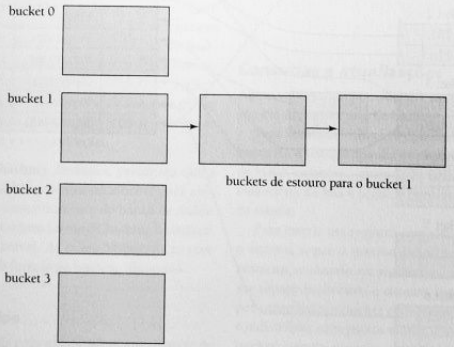


Figura 12.22 Encadeamento de estouro em uma estrutura de hash.

gum outro bucket no conjunto inicial de buckets B. Uma política é usar o próximo bucket (em ordem cíclica) que possui espaço. Essa política é chamada *sonda linear*. Outras políticas, como o cálculo de outras funções de hash, também são usadas. O hashing aberto tem sido usado para construir tabelas de símbolos para compiladores e assemblers, mas o hashing fechado é preferível para sistemas de banco de dados. O motivo é que a exclusão sob o hashing aberto é trabalhosa. Normalmente, compiladores e assemblers realizam apenas operações de pesquisa e inserção em suas tabelas de símbolos. Porém, em um sistema de banco de dados, é importante poder lidar com a exclusão tão bem quanto a inserção. Assim, o hashing aberto tem importância secundária na implementação do banco de dados.

Uma desvantagem importante da forma de hashing que descrevemos é que precisamos escolher a função de hash quando implementamos o sistema, e ela não pode ser mudada facilmente depois disso, se o arquivo sendo indexado crescer ou diminuir. Como a função h mapeia os valores de chave de busca a um conjunto fixo B dos endereços de bucket, desperdiçamos espaço se B se tornar grande para lidar com o crescimento futuro do arquivo. Se B for muito pequeno, os buckets contêm registros de muitos valores diferentes de chave de busca, e pode haver estouros de bucket. Com o crescimento do arquivo, o desempenho sofre. Estudaremos depois, na seção "Hashing dinâmico", como o número de buckets e a função de hash pode ser alterada dinamicamente.

Índices de hash

O hashing pode ser usado não apenas para organização de arquivo, mas também para a criação da estrutura de índice. Um **índice de hash** organiza as chaves de busca, com seus ponteiros associados em uma estrutura de arquivo de hash. Construímos um índice de hash da seguinte maneira. Aplicamos uma função de hash sobre uma chave de busca para identificar um bucket, e armazenamos a chave e seus ponteiros associados no bucket (ou em buckets de estouro). A Figura 12.23 mostra um índice de hash secundário sobre o arquivo *conta*, para a chave de busca *número_conta*. A função de hash na figura calcula a soma dos dígitos do número de conta módulo 7. O índice de hash possui sete buckets, cada um com tamanho 2 (índices realistas, naturalmente, teriam tamanhos de bucket muito maiores). Um dos buckets tem três chaves mapeadas, de modo que possui um

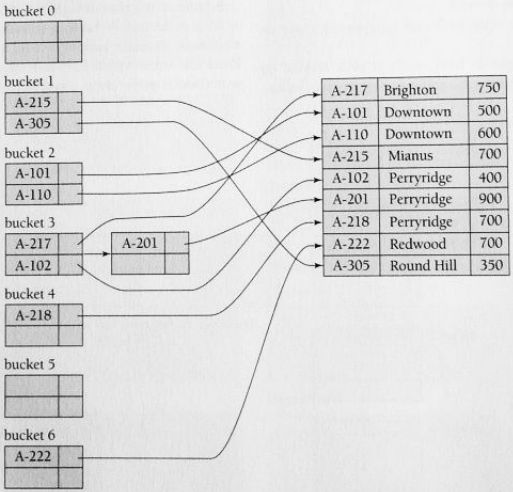


Figura 12.23 Índice de hash sobre a chave de busca *número_conta* do arquivo *conta*.

bucket de estouro. Nesse exemplo, *número_conta* é uma chave primária para *conta*, de modo que cada chave de busca tem apenas um ponteiro associado. Em geral, vários ponteiros podem estar associados a cada chave.

Usamos o termo *índice de hash* para indicar as estrutura de arquivo além de índices de hash secundários. Estritamente falando, os índices de hash são apenas estruturas de índice secundárias. Um índice de hash nunca é necessário como uma estrutura de índice agrupado, pois, se um arquivo já estiver organizado por hashing, não será necessária uma estrutura de índice de hash separada sobre ele. No entanto, como a organização de arquivo de hash oferece o mesmo acesso direto aos registros que a indexação, fazemos de conta que um arquivo organizado por hashing também possui um índice de hash agrupado sobre ele.

Hashing dinâmico

Como já vimos, a necessidade de consertar o conjunto B de endereços de bucket apresenta um problema sério com a técnica de hashing estático da seção anterior. A maioria dos bancos de dados se torna muito grande com o tempo. Se tivermos de usar o hashing estático para tal banco de dados, temos três classes de opções:

1. Escolha uma função de hash com base no tamanho do arquivo atual. Essa opção resultará na diminuição de desempenho quando o banco de dados crescer.
2. Escolha uma função de hash com base no tamanho antecipado do arquivo em algum ponto no futuro. Embora a diminuição de desempenho seja evitada, uma quantidade significativa de espaço pode ser desperdiçada inicialmente.
3. Reorganize periodicamente a estrutura de hash em resposta ao crescimento do arquivo. Tal reorganização envolve escolha de uma nova função de hash, recalculando a função de hash em cada registro do arquivo, e gerando novas atribuições de bucket. Essa reorganização é uma operação maciça e demorada. Além do mais, é necessário proibir o acesso ao arquivo durante a reorganização.

Várias técnicas de **hashing dinâmico** permitem que a função de hash seja modificada dinamicamente para acomodar o crescimento ou encolhimento do banco de dados. Nesta seção, descrevemos uma forma de hashing dinâmico, chamada **hashing extensível**. As notas bibliográficas contêm referências a outras formas de hashing dinâmico.

Estrutura de dados

O hashing extensível lida com as mudanças no tamanho do banco de dados, dividindo e unindo os buckets enquanto o banco de dados cresce e encurta. Como resultado, a eficiência do espaço é mantida. Além do mais, como a reorganização é realizada somente em um bucket de cada vez, a sobrecarga resultante no desempenho é aceitavelmente baixa.

Com o hashing extensível, escolhemos uma função de hash h com as propriedades desejáveis de uniformidade e aleatoriedade. Porém, essa função de hash gera valores por um intervalo relativamente grande – a saber, inteiros binários de b bits. Um valor típico de b é 32.

Não criamos um bucket para cada valor de hash. Na realidade, 2^{32} é mais de 4 bilhões, e essa quantidade é excessiva para quase tudo, menos para bancos de dados grandes. Em vez disso, criamos buckets por demanda, quando registros são inseridos no arquivo. Não usamos os b bits inteiros do valor de hash inicialmente. A qualquer ponto, usamos i bits, onde $0 \le i \le b$. Esses i bits são usados como um deslocamento para uma tabela adicional de endereços de bucket. O valor de i cresce e encurta com o tamanho do banco de dados.

A Figura 12.24 mostra uma estrutura geral de hash extensível. O i que aparece acima da tabela de endereços de bucket na figura indica que i bits do valor de hash $h(K)$ são necessários para determinar o bucket correto para K. Esse número, naturalmente, mudará quando o arquivo crescer. Embora i bits sejam necessários para encontrar a entrada correta na tabela de endereços de bucket, várias entradas de tabela consecutivas podem apontar para o mesmo bucket. Todas essas entradas terão um prefixo de hash comum, mas o tamanho desse prefixo pode ser menor que i. Portanto, associamos a cada bucket um inteiro dando o tamanho do prefixo de hash comum. Na Figura 12.24, o inteiro associado ao bucket j aparece como i_j. O número de entradas da tabela de endereços de bucket que apontam para o bucket j é

$$2^{(i-i_j)}$$

Consultas e atualizações

Agora veremos como realizar a pesquisa, inserção e exclusão em uma estrutura de hash extensível.

Para localizar o bucket contendo o valor de chave de busca K_j, o sistema apanha os primeiros i bits de alta ordem de $h(K_j)$, examina a entrada de tabela correspondente para essa string de bits e segue o ponteiro de bucket na entrada da tabela.

Para inserir um registro com valor de chave de busca K_j, o sistema segue o mesmo procedimento de antes, para a pesquisa, acabando em algum bucket – digamos, j. Se houver espaço no bucket, o sistema insere o registro nele . Se, por outro lado, o bucket estiver cheio, ele precisa dividi-lo e redistribuir os registros atuais, mais o novo. Para dividir o bucket, o sistema precisa primeiro determinar pelo valor de hash se ele precisa aumentar o número de bits utilizados.

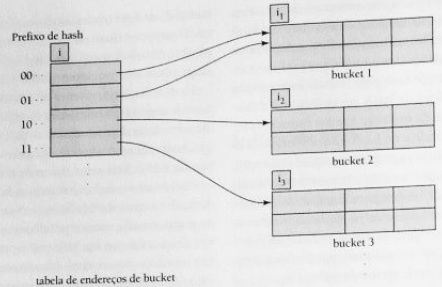


Figura 12.24 Estrutura de hash geral extensível.

- Se $i = i_j$, somente uma entrada na tabela de endereços de bucket aponta para o bucket j. Portanto, o sistema precisa aumentar o tamanho da tabela de endereços de bucket de modo que ela possa incluir ponteiros para os dois buckets que resultam da divisão do bucket j. Ele faz isso considerando um bit adicional do valor de hash. Ele incrementa o valor de i em 1, dobrando o tamanho da tabela de endereços de bucket. Ele substitui cada entrada por duas entradas, ambas contendo o mesmo ponteiro da entrada original. Agora, duas entradas na tabela de endereços de bucket apontam para o bucket j. O sistema aloca um novo bucket (bucket z) e define a segunda entrada para que aponte para o novo bucket. Ele define i_j e i_z como i. Em seguida, ele calcula novamente o hash de cada registro no bucket j e, dependendo dos i primeiros bits (lembre-se de que o sistema somou 1 a i), ele o mantém no bucket j ou o aloca ao bucket recém-criado.

  O sistema agora tenta novamente inserir o novo registro. Normalmente, a tentativa terá sucesso. Porém, se todos os registros no bucket j, além do novo registro, tiverem o mesmo prefixo de valor de hash, será preciso dividir o bucket novamente, pois todos os registros no bucket j e o novo registro são atribuídos ao mesmo bucket. Se a função de hash tiver sido cuidadosamente escolhida, é pouco provável que uma única inserção exija que o bucket seja dividido mais de uma vez, a menos que haja uma grande quantidade de registros com a mesma chave de busca. Se todos os registros no bucket j tiverem o mesmo valor de chave de busca, nenhuma quantidade de divisão ajudará. Nesses casos, os buckets de estouro são usados para armazenar os registros, como no hashing estático.
- Se $i > i_j$, então mais de uma entrada na tabela de endereços de bucket aponta para o bucket j. Assim, o sistema pode dividir o bucket j sem aumentar o tamanho da tabela de endereços de bucket. Observe que todas as entradas que apontam para o bucket j correspondem a prefixos de hash que têm o mesmo valor nos i_j bits mais à esquerda. O sistema aloca um novo bucket (bucket z) e define i_j e i_z com o valor que resulta da soma de 1 ao valor original de i_j. Em seguida, o sistema precisa ajustar as entradas na tabela de endereços de bucket que anteriormente apontava para o bucket j. (Observe que, com o novo valor para i_j, nem todas as entradas correspondem aos prefixos de hash que têm o mesmo valor nos i_j bits mais à esquerda.) O sistema deixa a primeira metade das entradas como estavam (apontando para o bucket j) e define todas as entradas restantes para que apontem para o bucket recém-criado (bucket z). Em seguida, como no caso anterior, o sistema recalcula o hash de cada registro no bucket j e o aloca ao bucket j ou ao bucket z recém-criado.

  O sistema, então, tenta inserir novamente. No caso improvável de nova falha, ele aplica um dos dois casos, $i = i_j$ ou $i > i_j$, como for mais apropriado.

Observe que, nos dois casos, o sistema precisa recalcular a função de hash apenas sobre os registros do bucket j.

Para excluir um registro com valor de chave de busca K_j, o sistema segue o mesmo procedimento de pesquisa anterior, acabando em algum bucket – digamos, j. Ele remove a chave de busca do bucket e o registro do arquivo. O bucket também é removido se ficar vazio. Observe que, nesse ponto, vários buckets podem ser unidos, e o tamanho da tabela de endereços de bucket pode ser reduzido ao meio. O procedimento para decidir quais buckets podem ser unidos e como uni-los fica para você como um exercício. As condições sob as quais a tabela de endereços de bucket pode ter o tamanho reduzido também ficam como um exercício. Ao contrário da união de buckets, a mudança do tamanho da

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Figura 12.25 Exemplo de arquivo de *conta*.

tabela de endereços de bucket é uma operação um tanto dispendiosa se a tabela for grande. Portanto, pode valer a pena reduzir o tamanho da tabela de endereços de bucket somente se o número de buckets reduzir muito.

Nosso arquivo de exemplo *conta* da Figura 12.25 ilustra a operação de inserção. Os valores de hash de 32 bits sobre *nome_agência* aparecem na Figura 12.26. Suponha que, inicialmente, o arquivo esteja vazio, como na Figura 12.27. Inserimos os registros um por um. Para ilustrar todos os recursos do hashing extensível em uma estrutura pequena, faremos a suposições não realista de que um bucket só pode manter dois registros.

Inserimos o registro (A-217, Brighton, 750). A tabela de endereços de bucket contém um ponteiro para um bucket, e o sistema insere o registro. Em seguida, inserimos o registro (A-101, Downtown, 500). O sistema também coloca esse registro no único bucket de nossa estrutura.

Quando tentamos inserir o próximo registro (A-110, Downtown, 600), descobrimos que o bucket está cheio. Como $i = i_0$, precisamos aumentar o número de bits que usamos a partir do valor de hash. Agora usamos 1 bit, permitindo-nos $2^1 = 2$ buckets. Esse aumento no número de bits exige dobrar o tamanho da tabela de endereços de bucket para duas entradas. O sistema divide o bucket, colocando no novo bucket aqueles registros cuja chave de busca possui um valor de hash começando com 1, e deixando no bucket original os outros registros. A Figura 12.28 mostra o estado de nossa estrutura após a divisão.

Em seguida, inserimos (A-215, Mianus, 700). Como o primeiro bit de h(Mianus) é 1, temos de inserir esse registro no bucket apontado pela entrada "1" na tabela de endereços de bucket. Mais uma vez, encontramos o bucket cheio e $i = i_1$. Aumentamos o número de bits que usamos a partir do hash para 2. Esse aumento no número de bits exige a dupli-

nome_agência	h(*nome_agência*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

Figura 12.26 Função de hash para *nome_agência*.

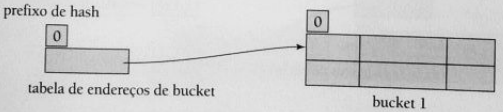


Figura 12.27 Estrutura de hash extensível inicial.

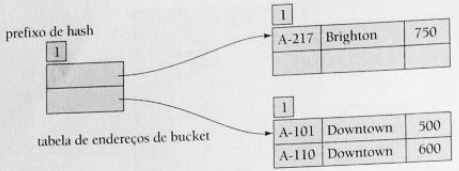


Figura 12.28 Estrutura de hash após três inserções.

cação do tamanho da tabela de endereços de bucket para quatro entradas, como na Figura 12.29. Como o bucket da Figura 12.28 para o prefixo de hash 0 não foi dividido, as duas entradas da tabela de endereços de bucket de 00 e 01 apontam para esse bucket.

Para cada registro no bucket da Figura 12.28 para o prefixo de hash 1 (o bucket sendo dividido), o sistema examina os 2 primeiros bits do valor de hash para determinar qual bucket da nova estrutura deverá mantê-lo.

Em seguida, inserimos (A-102, Perryridge, 400), que entra no mesmo bucket de Mianus. A inserção a seguir, de (A-201, Perryridge, 900), resulta em um estouro de bucket, levando a um aumento no número de bits e a duplicação do tamanho da tabela de endereços de bucket. A inserção do terceiro registro de Perryridge, (A-218, Perryridge, 700), leva a outro estouro. Porém, esse estouro não pode ser tratado pelo aumento do número de bits, pois existem três registros exatamente com o mesmo valor de hash. Logo, o sistema usa um bucket de estouro, como na Figura 12.30.

Continuamos dessa maneira até termos inserido todos os registros de *conta* da Figura 12.25. A estrutura resultante aparece na Figura 12.31.

Hashing estático versus hashing dinâmico

Agora, examinamos as vantagens e desvantagens do hashing extensível, em comparação com o hashing estático. A vantagem principal do hashing extensível é que o desempenho não diminui quando o arquivo aumenta. Além do mais, existe uma sobrecarga de espaço mínima. Embora a tabela de endereços de bucket gere uma sobrecarga adicional, ela contém um ponteiro para cada valor de hash para o tamanho de prefixo atual. Essa tabela, portanto, é pequena. A principal economia de espaço do hashing extensível em relação a outras formas de hashing é que nenhum bucket precisa ser reservado para crescimento futuro; em vez disso, os buckets podem ser alocados dinamicamente.

Uma desvantagem do hashing extensível é que a pesquisa envolve um nível de indireção adicional, pois o sistema precisa acessar a tabela de endereços de bucket antes de acessar o próprio bucket. Essa referência extra possui apenas um efeito secundário sobre o desempenho. Embora as estruturas de hash que discutimos na seção "Hashing estático" não tenham esse nível de indireção extra, elas perdem sua vantagem secundária no desempenho quando se tornam cheias.

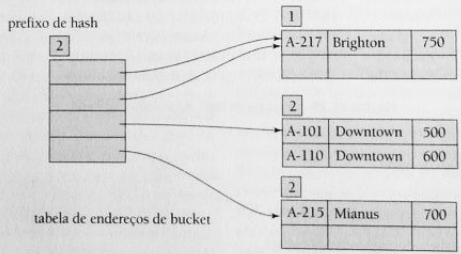


Figura 12.29 Estrutura de hash após quatro inserções.

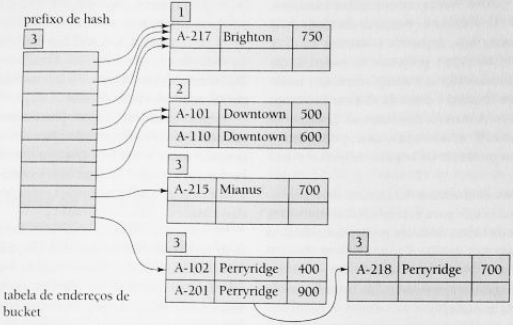


Figura 12.30 Estrutura de hash após sete inserções.

Assim, o hashing extensível parece ser uma técnica altamente atraente, desde que estejamos dispostos a aceitar a complexidade adicional envolvida em sua implementação. As notas bibliográficas referenciam descrições mais detalhadas da implementação de hashing extensível.

As notas bibliográficas também oferecem referências a outra forma de hashing dinâmico, chamado **hashing linear**, que evita o nível de indireção extra associado ao hashing extensível, ao custo possível de mais buckets de estouro.

Comparação de indexação ordenada e hashing

Vimos vários esquemas de indexação ordenada e vários esquemas de hashing. Podemos organizar arquivos de registros como arquivos ordenados, usando a organização sequencial indexada ou organizações de árvore B$^+$. Como alternativa, podemos organizar os arquivos usando o hashing. Finalmente, podemos organizá-los como arquivos de heap, em que os registros não são ordenados de qualquer modo em particular.

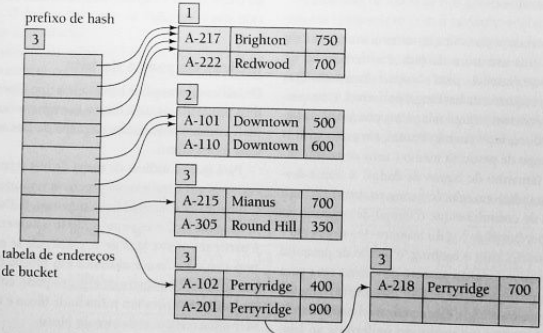


Figura 12.31 Estrutura de hash extensível para o arquivo *conta*.

Cada esquema possui vantagens em certas situações. Um implementador de sistema de banco de dados poderia oferecer muitos esquemas, deixando a decisão final de quais esquemas utilizar para o projetista do banco de dados. Porém, essa técnica exige que o implementador escreva mais código, aumentando o custo do sistema e o espaço que o sistema ocupa. A maioria dos sistemas de banco de dados admite árvores B+ e, adicionalmente, pode admitir alguma forma de organização de arquivo de hash ou índices de hash.

Para fazer uma escolha sensata das técnicas de organização de arquivo e indexação para uma relação, o implementador ou projetista de banco de dados precisa considerar as seguintes questões:

- O custo da reorganização periódica da organização de índice ou hash é aceitável?
- Qual é a freqüência relativa de inserção ou exclusão?
- É desejável otimizar o tempo médio de acesso às custas de aumentar o tempo de acesso no pior caso?
- Que tipos de consultas os usuários provavelmente realizam?

Já examinamos as três primeiras questões, primeiro em nossa revisão dos méritos relativos de técnicas de indexação específicas, e novamente em nossa discussão sobre técnicas de hashing. A quarta questão, o tipo de consulta esperado, é crítica para a escolha da indexação ordenada ou hashing.

Se a maioria das consultas for da forma

select $A_1, A_2, \ldots, A_n$
from r
where $A_i = c$

então, para processar a consulta, o sistema realizará uma pesquisa sobre uma estrutura de índice ordenado ou de hashing para os atributos A_i, para o valor c. Para consultas dessa forma, um esquema de hashing é preferível. Uma pesquisa de índice ordenado exige tempo proporcional ao log do número de valores em r para A_i. Porém, em uma estrutura de hash, o tempo de pesquisa médio é uma constante independente do tamanho do banco de dados. A única desvantagem de um índice em relação a uma estrutura de hash para essa forma de consulta é que o tempo de pesquisa no pior caso é proporcional ao log do número de valores em r para A_i. Ao contrário, para o hashing, o tempo de pesquisa no pior caso é proporcional ao número de valores em r para A_i. Porém, o tempo de pesquisa no pior caso provavelmente não ocorrerá com o hashing, e este é preferível nesse caso.

As técnicas de índice ordenado são preferíveis ao hashing em casos em que a consulta especifica um intervalo de valores. Essa consulta tem a seguinte forma:

select $A_1, A_2, \ldots, A_n$
from r
where $A_i \leq c_2$ and $A_i \geq c_1$

Em outras palavras, a consulta anterior localiza todos os registros com valores de A_i entre c_1 e c_2.

Vamos considerar como processamos essa consulta usando um índice ordenado. Primeiro, realizamos uma pesquisa sobre o valor c_1. Quando tivermos encontrado o bucket para o valor c_1, seguimos a cadeia de ponteiros no índice para ler o próximo bucket em ordem, e continuamos dessa maneira até alcançarmos c_2.

Se, em vez de um índice ordenado, tivermos uma estrutura de hash, podemos realizar uma pesquisa sobre c_1 e podemos localizar o bucket correspondente – mas, em geral, não é fácil determinar o próximo bucket que precisa ser examinado. A dificuldade surge porque uma boa função de hash atribui valores aleatoriamente aos buckets. Assim, não existe uma noção simples de "próximo bucket na ordem classificada". O motivo de não podermos encadear buckets em ordem classificada sobre A_i é que cada bucket recebe muitos valores de chave de busca. Como os valores são espalhados aleatoriamente pela função de hash, os valores no intervalo especificado provavelmente estarão espalhados por muitos ou todos os buckets. Portanto, temos de ler todos os buckets para encontrar as chaves de busca solicitadas.

Normalmente, projetista escolherá a indexação ordenada, a menos que se saiba com antecedência que as consultas de intervalo serão pouco freqüentes, quando o hashing seria escolhido. As organizações de hash são particularmente úteis para arquivos temporários criados durante o processamento da consulta, se as pesquisas baseadas em um valor de chave forem exigidas, mas nenhuma consulta de intervalo será realizada.

Índices de mapa de bits

Os índices de mapa de bits são um tipo especializado de índice projetado para facilitar a consulta sobre chaves múltiplas, embora cada índice de mapa de bits seja baseado em uma única chave.

Para que os índices de mapa de bits sejam utilizados, os registros em uma relação precisam ser numerados seqüencialmente, começando com, digamos, 0. Dado um número n, deve ser fácil recuperar o registro numerado com n. Isso é particularmente fácil de se conseguir se os registros tiverem tamanho fixo e alocados em blocos consecutivos de um arquivo. O número de registro pode, então, ser traduzido facilmente para um número de bloco e um número que identifica o registro dentro do bloco.

Considere uma relação r, com um atributo A que pode assumir apenas um dentre um pequeno número de valores

(por exemplo, 2 a 20). Por exemplo, uma relação *info_cliente* pode ter um atributo *sexo*, que pode assumir apenas os valores m (masculino) ou f (feminino). Outro exemplo seria um atributo *nível_renda*, em que a renda foi dividida em 5 níveis: L1: 0-9999, L2: 10.000-19.000, L3: 20.00-39.999, L4: 40.000-74.999 e L5: 75.000-∞. Aqui, os dados brutos podem assumir muitos valores, mas um analista de dados dividiu os valores em um pequeno número de intervalos para simplificar a análise desses dados.

Estrutura de índice de mapa de bits

Um **mapa de bits** é simplesmente um array de bits. Em sua forma mais simples, um **índice de mapa de bits** sobre o atributo A da relação r consiste em um mapa de bits para cada valor que A pode assumir. Cada mapa de bits possui tantos bits quanto o número de registros na relação. O i-ésimo bit do mapa de bits para o valor v_j é definido como 1 se o registro numerado com i tiver o valor v_j para o atributo A. Todos os outros bits do mapa de bits são definidos como 0.

Em nosso exemplo, existe um mapa de bits para o valor m e um para f. O i-ésimo bit do mapa de bits para m é definido como 1 se o valor de *sexo* do registro numerado com i for m. Todos os outros bits do mapa de bits para m são definidos como 0. De modo semelhante, o mapa de bits para f tem o valor 1 para os bits correspondentes aos registros com o valor f para o atributo *sexo*. Todos os outros bits têm o valor 0. A Figura 12.32 mostra um exemplo dos índices de mapa de bits em uma relação *info_cliente*.

Agora, vejamos quando os mapas de bit são úteis. A forma mais simples de obter todos os registros com valor m (ou valor f) seria simplesmente ler todos os registros da relação e selecionar aqueles registros com valor m (ou f, respectivamente). O índice do mapa de bits não ajuda realmente a agilizar essa seleção.

De fato, os índices de mapa de bits são úteis para seleções principalmente quando existem seleções sobre chaves múltiplas. Suponha que criemos um índice de mapa de bits sobre o atributo *nível_renda*, que descrevemos anteriormente, além do índice de mapa de bits sobre *sexo*.

Considere agora uma consulta que seleciona mulheres com renda no intervalo de 10.000-19.999. Essa consulta pode ser expressa como $\sigma_{sexo=f \wedge nivel\_renda=L2}(r)$. Para avaliar essa seleção, apanhamos os mapas de bits para o valor f de sexo e o mapa de bits para o valor *L2* de *nível_renda*, e realizamos uma **interseção** (and lógico) dos dois mapas de bits. Em outras palavras, calculamos um novo mapa de bits em que o bit i tem valor 1 se o i-ésimo bit dos dois mapas de bit forem 1; caso contrário, ele tem um valor 0. No exemplo da Figura 12.32, a interseção do mapa de bits para *sexo* = f (01101) e o mapa de bits para *nível_renda* = *L2* (01000) gera o mapa de bits 01000.

Como o primeiro atributo pode assumir 2 valores, e o segundo pode assumir 5 valores, esperaríamos que apenas cerca de 1 em 10 registros, na média, satisfaça uma condição combinada sobre os dois atributos. Se houver outras condições, a fração dos registros que satisfazem todas as condições provavelmente será muito pequeno. O sistema pode, então, calcular o resultado da consulta localizando todos os bits com valor 1 no mapa de bits da interseção e apanhando os registros correspondentes. Se a fração for grande, a varredura da relação inteira continuaria sendo a alternativa mais barata.

Outro uso importante dos mapas de bits é contar o número de tuplas satisfazendo determinada seleção. Essas consultas são importantes para a análise de dados. Por exemplo, se quisermos descobrir quantas mulheres possuem um nível de renda *L2*, calculamos a interseção dos dois mapas de bits e depois contamos o número de bits que são 1 no mapa de bits da interseção. Assim, podemos chegar ao resultado a partir do índice de mapa de bits, sem sequer acessar a relação.

Os índices de mapa de bits geralmente são muito pequenos em comparação com o tamanho real da relação. Os registros normalmente têm pelo menos dezenas de bytes a centenas de bytes, enquanto um único bit representa um registro em um mapa de bits. Assim, o espaço ocupado por um único mapa de bits normalmente é menor do que 1% do espaço ocupado pela relação. Por exemplo, se o tamanho do registro para determinada relação for de 100 bytes, en-

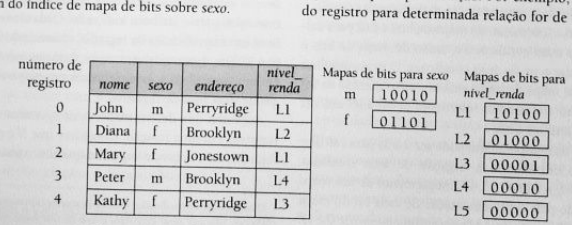


Figura 12.32 Índices de mapa de bits na relação *info_cliente*.

tão o espaço ocupado por um único mapa de bits seria 1/8 do 1% do espaço ocupado pela relação. Se um atributo *A* da relação puder assumir apenas um dentre 8 valores, um índice de mapa de bits sobre o atributo *A* consistiria em 8 mapas de bits, que juntos ocupariam apenas 1% do tamanho da relação.

A exclusão de registros cria lacunas na seqüência de registros, pois o deslocamento de registros (ou números de registro) para preencher as lacunas seria extremamente dispendioso. Para reconhecer os registros excluídos, podemos armazenar um **mapa de bits de existência**, em que o bit *i* é 0 se o registro *i* não existir, e 1 se ele existir. Veremos a necessidade da existência de mapas de bits na próxima seção. A inserção de registros não deverá afetar a numeração de seqüência de outros registros. Portanto, podemos realizar a inserção acrescentando registros ao final do arquivo ou substituindo os registros excluídos.

Implementação eficiente de operações de mapa de bits

Podemos calcular a interseção dos dois mapas de bits com facilidade usando um loop **for**: a *i*-ésima iteração do loop calcula o **and** dos *i*-ésimos bits dos dois mapas de bits. Podemos agilizar bastante o cálculo da interseção usando instruções **and** bit a bit, aceitas pela maioria dos conjuntos de instruções de computador. Uma palavra (*word*) normalmente consiste em 32 ou 64 bits, dependendo da arquitetura do computador. Uma instrução **and** bit a bit apanha duas palavras como entrada e gera uma palavra em que cada bit é o **and** lógico dos bits nas posições correspondentes das palavras de entrada. O que é importante observar é que uma única instrução **and** bit a bit pode calcular a interseção de 32 ou 64 bits *ao mesmo tempo*.

Se uma relação tivesse 1 milhão de registros, cada mapa de bits teria 1 milhão de bits, ou 128 kilobytes. Apenas 31.250 instruções são necessárias para calcular a interseção dos dois mapas de bits para nossa relação, considerando um tamanho de palavra de 32 bits. Assim, o cálculo das interseções de mapa de bits é uma operação extremamente rápida.

Assim como a interseção de mapa de bits é útil para calcular o **and** de duas condições, a união de mapa de bits é útil para calcular o **or** de duas condições. O procedimento para a união de mapa de bits é exatamente o mesmo da interseção, exceto que usamos instruções **or** bit a bit em vez de instruções **and** bit a bit.

A operação de complemento pode ser usada para calcular um predicado envolvendo a negação de uma condição, como **not** (*nível_renda* = *L1*). O complemento de um mapa de bits é gerado pela complementação de cada bit do mapa de bits (o complemento de 1 é 0, e o complemento de 0 é 1). Pode parecer que **not** (*nível_renda* = *L1*) pode ser implementado apenas calculando-se o complemento do mapa de bits para o nível de renda *L1*. Porém, se alguns registros tiverem sido excluídos, simplesmente calcular o complemento de um mapa de bits não é suficiente. Os bits correspondentes a tais registros seriam 0 no mapa de bits original, mas se tornariam 1 no complemento, embora os registros não existam. Um problema semelhante também surge quando o valor de um atributo é *nulo*. Por exemplo, se o valor de *nível_renda* fosse nulo, o bit seria 0 no mapa de bits original para o valor *L1*, e 1 no mapa de bits do complemento.

Para ter certeza de que os bits correspondentes aos registros excluídos sejam definidos como 0 no resultado, o mapa de bits do complemento precisa passar pela interseção com o mapa de bits de existência, de modo a desativar os bits para os registros excluídos. De modo semelhante, para lidar com valores nulos, o mapa de bits de complemento também precisa sofrer uma interseção com o complemento do mapa de bits para o valor *nulo*.[1]

A contagem do número de bits que são 1 em um mapa de bits pode ser feita rapidamente por uma técnica inteligente. Podemos manter um array com 256 entradas, em que a *i*-ésima entrada armazena o número de bits que são 1 na representação binária de *i*. Defina a contagem total inicialmente como 0. Apanhamos cada byte do mapa de bits, o usamos para indexar esse array e somamos a contagem armazenada à contagem total. O número de operações de adição seria 1/8 do número de tuplas, e assim o processo de contagem é muito eficiente. Um array grande (usando 2^{16} = 65.536 entradas), indexado por pares de bytes, ocasionaria um aumento de velocidade ainda maior, mas a um custo mais alto no armazenamento.

Mapas de bits e árvores B⁺

Os mapas de bits podem ser combinados com os índices normais de árvore B⁺ para relações em que alguns valores de atributo são extremamente comuns, e outros valores também ocorrem, mas com muito menos freqüência. Em uma folha de índice de árvore B⁺, para cada valor, normalmente manteríamos uma lista de todos os registros com esse valor para o atributo indexado. Cada elemento da lista seria um identificador de registro, consistindo em pelo menos 32 bits, e normalmente mais. Para um valor que ocorre em muitos registros, armazenamos um mapa de bits no lugar de uma lista de registros.

Suponha que determinado valor v_i ocorra em 1/16 dos registros em uma relação. Considere que *N* é o número de registros na relação, e suponha que um registro tenha um

1. O tratamento de predicados como é **desconhecido** causaria outras complicações, que em geral exigiriam o uso de um mapa de bits extra para acompanhar quais resultados de operação são desconhecidos.

número de 64 bits que o identifica. O mapa de bits só precisa de 1 bit por registro, ou N bits no total. Ao contrário, a representação de lista exige 64 bits por registro em que o valor ocorre, ou $64 * N/16 = 4N$ bits. Assim, um mapa de bits é preferível para representar a lista de registros para o valor v_i. Em nosso exemplo (com um identificador de registro de 64 bits), se menos de 1 em 64 registros tiverem determinado valor, a representação de lista é preferível para identificar registros com esse valor, pois utiliza menos bits do que a representação de mapa de bits. Se mais de 1 em 64 registros tiverem esse valor, a representação de mapa de bits será preferível.

Assim, os mapas de bits podem ser usados como um mecanismo de armazenamento compactado nos nos de folha das árvores B^+, para os valores que ocorrem com muita freqüência.

Definição de índice na SQL

O padrão SQL não oferece um meio para o usuário ou administrador do banco de dados controlar quais índices são criados e mantidos no sistema de banco de dados. Os índices não são necessários para exatidão, pois são estruturas de dados redundantes. Porém, os índices são importantes para o processamento eficiente de transações, incluindo transações e consultas de atualização. Os índices também são importantes para a imposição eficiente das restrições de integridade. Por exemplo, as implementações típicas impõem uma declaração de chave (Capítulo 4) criando um índice com a chave declarada como chave de busca do índice.

Em geral, um sistema de banco de dados pode decidir automaticamente quais índices devem ser criados. Porém, devido ao custo de espaço dos índices, bem como o efeito dos índices sobre o processamento de atualização, não é fácil fazer automaticamente as escolhas certas sobre quais índices manter. Portanto, a maioria das implementações da SQL oferece ao programador o controle sobre a criação e a remoção de índices via comandos da linguagem de definição de dados.

Ilustramos a sintaxe desses comandos sem seguida. Embora a sintaxe apresentada seja muito usada e aceita por muitos sistemas de banco de dados, ela não faz parte do padrão SQL:1999. Os padrões da SQL (até o SQL:1999, pelo menos) não admitem o controle do esquema do banco de dados, e se restringiram ao esquema lógico do banco de dados.

Criamos um índice com o comando **create index**, que tem o seguinte formato:

create index <nome-índice> on
<nome-relação> (<lista-atributos>)

A *lista-atributos* é uma lista dos atributos das relações que formam a chave de busca para o índice.

Para definir um nome de índice *índice_agência* sobre a relação *agência* com *nome_agência* como chave de busca, escrevemos

create index *índice_agência* **on** *agência* (*nome_agência*)

Se quisermos declarar que a chave de busca é uma chave candidata, acrescentamos o atributo **unique** à definição do índice. Assim, o comando

create unique index *índice_agência* **on** *agência* (*nome_agência*)

declara *nome_agência* como uma chave candidata para *agência*. Se, no momento em que entrarmos com o comando **create unique index**, *nome_agência* não for uma chave candidata, o sistema mostrará uma mensagem de erro, e a tentativa de criar o índice falhará. Se a tentativa de criação de índice tiver sucesso, qualquer tentativa subseqüente para inserir uma tupla que infrinja a declaração de chave falhará. Observe que o recurso **unique** é redundante se o sistema de banco de dados admitir a declaração **unique** do padrão SQL.

Muitos sistemas de banco de dados também oferecem um meio de especificar o tipo de índice a ser usado (como árvore B^+ ou hashing). Alguns sistemas de banco de dados também permitem que um dos índices em uma relação seja declarado como agrupado; o sistema, então, armazena a relação classificada pela chave de busca do índice agrupado.

O nome do índice que especificamos para um índice é necessário para a remoção de um índice. O comando **drop index** tem este formato:

drop index <*nome-índice*>

Resumo

- Muitas consultas referenciam apenas uma pequena proporção dos registros em um arquivo. Para reduzir a sobrecarga na pesquisa por esses registros, podemos construir *índices* para os arquivos que armazenam o banco de dados.
- Os arquivos seqüenciais indexados são um dos esquemas de índice mais antigos usados nos sistemas de banco de dados. Para permitir a rápida recuperação de registros na ordem da chave de busca, os registros são armazenados seqüencialmente, e aqueles fora de ordem são encadeados. Para permitir o acesso aleatório rápido, usamos uma estrutura de índice.
- Existem dois tipos de índices que podemos usar: índices densos e índices esparsos. Os índices densos contêm en-

tradas para cada valor de chave de busca, enquanto os índices esparsos contêm entradas somente para os valores da chave de busca.

- Se a ordem de classificação de uma chave de busca combinar com a ordem de classificação de uma relação um índice na chave de busca é chamado de *índice agrupado*. Os outros índices são denominados *índices não agrupados* ou *secundários*. Os índices secundários melhoram o desempenho das consultas que usam chaves de busca diferentes da chave de busca para o índice agrupado. Porém, eles impõem uma sobrecarga na modificação do banco de dados.
- A principal desvantagem da organização de arquivo sequencial indexada é que o desempenho diminui à medida que o arquivo cresce. Para contornar essa deficiência, podemos usar um *índice de árvore B*$^+$.
- Um índice de árvore B$^+$ tem a forma de uma árvore *balanceada*, em que cada caminho da raiz da árvore até uma folha da árvore tem o mesmo tamanho. A altura de uma árvore B$^+$ é proporcional ao logaritmo de base N do número de registros na relação, em que cada nó não-folha armazena N ponteiros; o valor de N normalmente fica em torno de 50 a 100. As árvores B$^+$ são muito mais curtas do que outras estruturas de árvore binária balanceada, como árvores AVL, e por isso exigem menos acesso ao disco para localizar registros.
- A pesquisa nas árvores B$^+$ são diretas e eficientes. Porém, inserção e exclusão são um pouco mais complicadas, mas ainda eficientes. O número de operações exigidas para pesquisa, inserção e exclusão nas árvores B$^+$ é proporcional ao logaritmo de base N do número de registros na relação, em que cada nó não de folha armazena N ponteiros.
- Podemos usar árvores B$^+$ para indexar um arquivo contendo registros, além de organizar registros em um arquivo.
- Índices de árvore B são semelhantes aos índices de árvore B$^+$. A principal vantagem de uma árvore B é que a árvore B elimina o armazenamento redundante dos valores de chave de busca. As principais desvantagens são a complexidade geral e o fanout reduzido para determinado tamanho de nó. Os projetistas de sistemas quase unanimemente preferem os índices de árvore B$^+$ aos índices de árvore B na prática.
- Organizações de arquivo seqüenciais exigem uma estrutura de índice para localizar dados. As organizações de arquivo baseadas em hashing, ao contrário, nos permitem localizar o endereço de um item de dados diretamente pelo cálculo de uma função sobre o valor de chave de busca do registro desejado. Como não sabemos durante o projeto exatamente quais valores de chave de busca serão armazenados no arquivo, uma boa função de hash para escolher é aquela que atribui valores de chave de busca aos buckets de modo que a distribuição seja uniforme e aleatória.
- O *hashing estático* utiliza funções de hash em que o conjunto de endereços de bucket é fixo. Essas funções de hash não podem acomodar com facilidade bancos de dados que se tornam muito grandes com o tempo. Existem várias *técnicas de hashing dinâmico*, que permitem a modificação da função de hash. Um exemplo é o *hashing extensível*, que lida com as mudanças no tamanho do banco de dados pela divisão e união de buckets à medida que o banco de dados cresce e encurta.
- Também podemos usar o hashing para criar índices secundários; esses índices são chamados *índices de hash*. Por conveniência de notação, assumimos que as organizações de arquivo possuem um índice de hash implícito sobre a chave de busca usada para o hashing.
- Índices ordenados, como árvores B$^+$ e índices de hash, podem ser usados para seleções com base em condições de igualdade envolvendo atributos isolados. Quando vários atributos estão envolvidos em uma condição de seleção, podemos realizar a interseção de identificadores de registro apanhados de vários índices.
- Os índices de mapa de bits oferecem uma representação bastante compacta para atributos de indexação com valores muito pouco distintos. As operações de interseção são extremamente rápidas nos mapas de bits, tornando-os ideais para o suporte a consultas sobre atributos multiplos.

Termos de revisão

- Tipos de acesso
- Tempo de acesso
- Tempo de inserção
- Tempo de exclusão
- Sobrecarga de espaço
- Índice ordenado
- Índice agrupado
- Índice primário
- Índice não agrupado
- Índice secundário
- Arquivo seqüencial indexado
- Registro/entrada de índice
- Índice denso
- Índice esparso
- Índice multinível
- Chave composta
- Varredura seqüencial
- Índice de árvore B$^+$
- Árvore balanceada
- Organização de arquivo de árvore B$^+$

- Índice de árvore B
- Hashing estático
- Organização de arquivo de hash
- Índice de hash
- Bucket
- Função de hash
- Estouro de bucket
- Distorção
- Hashing fechado
- Hashing dinâmico
- Hashing extensível
- Acesso por chave múltipla
- Índices sobre chaves múltiplas
- Índice de mapa de bits
- Operações de mapa de bits
  - Interseção
  - União
  - Complemento
  - Mapa de bits de existência

Exercícios práticos

12.1 Alguns índices agilizam o processamento da consulta; por que eles não poderiam ser mantidos em várias chaves de busca? Liste o máximo de respostas possível.

12.2 É possível, em geral, ter dois índices agrupados na mesma relação para diferentes chaves de busca? Explique sua resposta.

12.3 Construa uma árvore B^+ para o seguinte conjunto de valores de chave:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Suponha que a árvore esteja inicialmente vazia e os valores sejam acrescentados em ordem crescente. Construa árvores B^+ para os casos em que o número de ponteiros que caberão em um nó é o seguinte:

a. Quatro
b. Seis
c. Oito

12.4 Para cada árvore B^+ do Exercício 12.3, mostre a forma da árvore após cada uma das seguintes séries de operações:

a. Inserir 9.
b. Inserir 10.
c. Inserir 8.
d. Excluir 23.
e. Excluir 19.

12.5 Considere o esquema de redistribuição modificado para as árvores B^+ descritas na seção "Organização de arquivos de árvore B^{+}". Qual é a altura esperada da árvore como uma função de n?

12.6 Repita o Exercício prático 12.3 para uma árvore B.

12.7 Suponha que estejamos usando o hashing extensível sobre um arquivo que contém registros com os seguintes valores de chave de busca:

2, 3, 5, 7, 11, 17, 19, 23, 29, 31

Mostre a estrutura de hash extensível para esse arquivo se a função de hash for $h(x) = x \bmod 8$ e os buckets puderem manter três registros.

12.8 Mostre como a estrutura de hash extensível do Exercício prático 12.7 muda como resultado de cada uma das seguintes etapas:

a. Excluir 11.
b. Excluir 31.
c. Inserir 1.
d. Inserir 15.

12.9 Dê o pseudocódigo para a exclusão de entradas de uma estrutura de hash extensível, incluindo detalhes de quando e como unir os buckets. Não se preocupe em reduzir o tamanho da tabela de endereços de bucket.

12.10 Sugira um modo eficiente de testar se a tabela de endereços de bucket no hashing extensível pode ser reduzida em tamanho, armazenando uma contagem extra com a tabela de endereços de bucket. Dê os detalhes de como a contagem deve ser mantida quando os buckets são divididos, unidos ou excluídos.
(Nota: A redução do tamanho da tabela de endereços de bucket é uma operação dispendiosa, e as inserções subseqüentes podem fazer com que a tabela cresça novamente. Portanto, é melhor não reduzir o tamanho assim que for possível, mas, em vez disso, reduzir apenas se o número de entradas de índice se tornar pequeno em comparação com o tamanho da tabela de endereços de bucket.)

12.11 Considere a relação *conta* mostrada na Figura 12.25.

a. Construa um índice de mapa de bits sobre os atributos *nome_agência* e *saldo*, dividindo os valores de *saldo* em 4 intervalos: abaixo de 250, 250 até abaixo de 500, 500 até abaixo de 750 e 750 em diante.
b. Considere uma consulta que solicita todas as contas em Downtown com um saldo superior ou igual a 500. Esboce as etapas para responder a consulta, e mostre os mapas de bit finais e intermediários construídos para responder a consulta.

12.12 Suponha que você tenha uma relação com n_r tuplas em que uma árvore B^+ secundária deve ser construída.

a. Dê uma fórmula para o custo de construção do índice de árvore B* inserindo um registro de cada vez. Suponha que cada página mantenha uma média de f entrada, e que todos os níveis da árvore acima da folha estejam na memória.

b. Considerando um tempo de acesso ao disco de 10 milissegundos, qual é o custo da construção de índice em uma relação com 10 milhões de registros?

c. Sugira um modo mais eficiente de construir o índice *de baixo para cima*, construindo primeiro o nível de folha inteiro e depois níveis mais altos, um por vez. Suponha que você tenha uma função que possa classificar, de modo eficiente, um conjunto muito grande de registros, mesmo o conjunto seja maior do que pode caber na memória. (Esses algoritmos de ordenação são descritos mais adiante, na seção "Classificação" do Capítulo 13 e, considerando uma quantidade razoável de memória principal, eles têm um custo de aproximadamente uma operação de E/S por bloco.)

Exercícios

12.13 Quando é preferível usar um índice denso no lugar de um índice esparso? Explique sua resposta.

12.14 Qual é a diferença entre um índice agrupado e um índice secundário?

12.15 Para cada árvore B*+ do Exercício prático 12.3, mostre as etapas envolvidas nas seguintes consultas:

a. Encontrar registros com um valor de chave de busca igual a 11.

b. Encontrar registros com um valor de chave de busca entre 7 e 17, inclusive.

12.16 A solução apresentada na seção "Chaves de busca não exclusivas" para lidar com chaves de busca não exclusivas acrescentou um atributo extra à chave de busca. Que efeito essa mudança tem sobre a altura da árvore B*+?

12.17 Explique a distinção entre o hashing fechado e aberto. Discuta os méritos relativos de cada técnica nas aplicações de banco de dados.

12.18 Quais são os casos de estouro de bucket em uma organização de arquivo de hash? O que pode ser feito para reduzir a ocorrência de estouros de bucket?

12.19 Por que uma estrutura de hash não é a melhor escolha para uma chave de busca em que as consultas de intervalo são prováveis?

12.20 Suponha que haja uma relação $R(A,B,C)$, com um índice de árvore B+ com chave de busca (A,B).

a. Qual é o custo no pior caso para encontrar os registros que satisfazem $10 < A < 50$, usando esse índice, em termos do número de registros apanhados n_1 e a altura h da árvore?

b. Qual é o custo no pior caso para encontrar os registros que satisfazem $10 < A < 50 \wedge 5 < B < 10$ usando esse índice, em termos do número de registros n_2 que satisfazem essa seleção, bem como n_1 e h, definidos anteriormente?

c. Sob que condições sobre n_1 e n_2 o índice seria um modo eficiente de encontrar registros que satisfaçam $10 < A < 50 \wedge 5 < B < 10$?

12.21 Suponha que você tenha de criar um índice de árvore B+ sobre uma grande quantidade de nomes, em que o tamanho máximo de um nome pode ser muito grande (digamos, 40 caracteres) e, na média, o nome seja grande (digamos 10 caracteres). Explique como a compactação do prefixo pode ser usada para melhorar o fanout médio dos nós internos.

12.22 Por que os nós de folha de uma organização de arquivo de árvore B+ poderiam perder sua seqüencialidade? Sugira como a organização do arquivo pode ser reorganizada para restaurar a seqüencialidade?

12.23 Suponha que uma relação seja armazenada em uma organização de arquivo de árvore B+. Suponha que os índices secundários armazenassem identificadores de registro que são ponteiros para registros no disco.

a. Qual seria o efeito dos índices secundários se uma divisão de página ocorresse na organização do arquivo?

b. Qual seria o custo de atualizar todos os registros afetados em um índice secundário?

c. Como o uso da chave de busca da organização de arquivo como um identificador de registro lógico soluciona esse problema?

d. Qual é o custo extra devido ao uso de tais identificadores de registro lógico?

12.24 Mostre como calcular a existência de mapas de bits a partir de outros mapas de bits. Certifique-se de que sua técnica funciona mesmo na presença de valores nulos, usando um mapa de bits para o valor *nulo*.

12.25 Como a criptografia de dados afeta os esquemas de índice? Em particular, como ela poderia afetar os esquemas que tentam armazenar dados em ordem classificada?

12.26 Nossa descrição do hashing estático considera que uma grande faixa contígua de blocos de disco pode ser alocada a uma tabela de hash estática. Suponha que você possa alocar apenas C blocos contíguos. Sugira como implementar a tabela de hash, se ela puder ser maior do que C blocos. O acesso a um bloco ainda deverá ser eficiente.

Notas bibliográficas

As discussões sobre as estruturas de dados básicas na indexação e no hashing podem ser encontradas em Cormen *et al.* [1990]. Os índices de árvore B foram introduzidos inicialmente em Bayer [1972] e Bayer e McCreight [1972]. As árvores B^+ são discutidas em Comer [1979], Bayer e Unterauer [1977] e Knuth [1973]. As notas bibliográficas no Capítulo 16 oferecem referências à pesquisa sobre a permissão dos acessos concorrentes e atualizações em árvores B^+. Gray e Reuter [1993] oferecem uma boa descrição das questões na implementação de árvores B^+.

Várias estruturas alternativas de árvore e tipo árvore foram propostas. **Tries** são árvores cuja estrutura é baseada nos "dígitos" das chaves (por exemplo, um índice de marcador de dicionário, que tem uma entrada para cada letra). Essas árvores podem ser balanceadas no mesmo sentido das árvores B^+. Tries são discutidas por Ramesh *et al.* [1989], Orestein [1982], Litwin [1981] e Fredkin [1960]. O trabalho relacionado inclui as árvores B de Lomet [1981].

Knuth [1973] analisa uma grande quantidade de técnicas de hashing diferentes. Existem vários esquemas de hashing dinâmico. O hashing extensível foi introduzido por Fagin *et al.* [1979]. O hashing linear foi introduzido por Litwin [1978, 1980]. Uma comparação de desempenho com hashing extensível é fornecida por Rathi *et al.* [1990]. Uma alternativa dada por Ramakrishna e Larson [1989] permite a recuperação em um único acesso ao disco, ao preço de uma sobrecarga alta para uma pequena fração de modificações do banco de dados. O hashing particionado é uma extensão do hashing para vários atributos, e é explicado em Rivest [1976], Byrjgard [1976] e Burkhard [1979].

Vitter [2001] oferece um estudo abrangente sobre as estruturas e os algoritmos de dados na memória externa.

Os índices de mapa de bits, e suas variantes chamadas **índice de bits** e **índices de projeção**, são descritos em O'Neil e Quass [1997]. Eles foram introduzidos inicialmente no gerenciador de arquivos IBM Model 204, na plataforma AS 400. Eles oferecem aumentos de velocidade muito grandes em certos tipos de consultas, e hoje são implementados na maioria dos sistemas de banco de dados. A pesquisa recente sobre os índices de mapa de bits inclui Wu e Buchmann [1998], Chan e Ioannidis [1998], Chan e Ioannidis [1999] e Johnson [1999].

CAPÍTULO 13

Processamento da consulta

O **processamento da consulta** refere-se ao conjunto de atividades envolvidas na extração de dados de um banco de dados. As atividades incluem tradução de consultas em linguagens de banco de dados de alto nível para expressões que podem ser usadas no nível físico do sistema de arquivos, uma série de transformações de otimização da consulta e a avaliação real das consultas.

Visão geral

As etapas envolvidas no processamento de uma consulta aparecem na Figura 13.1. São elas:

1. Análise e tradução
2. Otimização
3. Avaliação

Antes que qualquer processamento de consulta possa começar, o sistema precisa traduzir a consulta para uma forma utilizável. Uma linguagem como SQL é adequada para o uso humano, mas não para a representação interna do sistema de uma consulta. Uma representação interna mais útil é aquela baseada na álgebra relacional estendida.

Assim, a primeira ação que o sistema precisa tomar no processamento da consulta é traduzir determinada consulta para a sua forma interna. Esse processo de tradução é semelhante ao trabalho realizado pelo analisador (parser) de um compilador. Ao gerar a forma interna da consulta, o analisador verifica a sintaxe da consulta do usuário e se os nomes de relação que aparecem na consulta são nomes de relações no banco de dados etc. O sistema constrói uma representação de árvore de análise da consulta, que pode traduzir para uma expressão da álgebra relacional. Se a con-

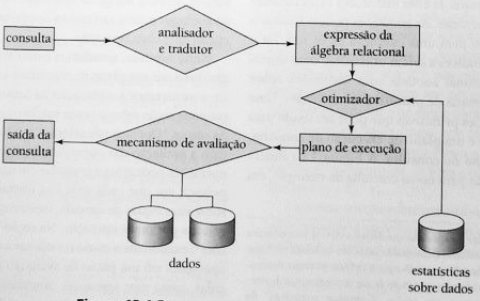


Figura 13.1 Etapas no processamento da consulta.

sulta foi expressa em termos de uma view, a fase de tradução também substitui todos os usos da view pela expressão da álgebra relacional que define a view.[1] A maioria dos textos sobre compilador explica a análise (veja nas notas bibliográficas).

Dada uma consulta, geralmente existem vários métodos para calcular a resposta. Por exemplo, vimos que, em SQL, uma consulta poderia ser expressa de várias maneiras diferentes. Cada consulta SQL pode ser traduzida para uma expressão da álgebra relacional em uma dentre várias maneiras. Além do mais, a representação da álgebra relacional de uma consulta especifica apenas parcialmente como avaliar uma consulta; normalmente, existem várias maneiras de avaliar as expressões da álgebra relacional. Como ilustração, considere a consulta

```
select saldo
from conta
where saldo < 2500
```

Essa consulta pode ser traduzida para uma das seguintes expressões da álgebra relacional:

- $\sigma_{saldo<2500}(\Pi_{saldo}(conta))$
- $\Pi_{saldo}(\sigma_{saldo<2500}(conta))$

Além do mais, podemos executar cada operação da álgebra relacional por um de vários algoritmos diferentes. Por exemplo, para implementar a seleção anterior, podemos pesquisar cada tupla em *conta* para encontrar tuplas com saldo menor que 2500. Se um índice de árvore B+ estiver disponível sobre o atributo *saldo*, podemos usar o índice em vez de localizar as tuplas.

Para especificar totalmente como avaliar uma consulta, precisamos não apenas oferecer a expressão da álgebra relacional mas também anotá-la com instruções especificando como avaliar cada operação. As anotações podem indicar o algoritmo a ser usado para uma operação específica, ou o índice específico ou índices a serem utilizados. Uma operação da álgebra relacional anotada com instruções sobre como avaliá-la é chamada de **primitiva de avaliação**. Uma seqüência de operações primitivas que pode ser usada para avaliar uma consulta é um **plano de execução de consulta**, ou **plano de avaliação de consulta**. A Figura 13.2 ilustra um plano de avaliação para nossa consulta de exemplo, em

Figura 13.2 Um plano de avaliação de consulta.

que um índice em particular (denotado na figura como "índice 1") é especificado para a operação de seleção. O **mecanismo de execução de consulta** apanha um plano de avaliação de consulta, executa esse plano e retorna as respostas à consulta.

Os diferentes planos de avaliação para determinada consulta podem ter diferentes custos. Não esperamos que os usuários escrevam suas consultas de modo a sugerir o plano de avaliação mais eficiente. Em vez disso, é responsabilidade do sistema construir um plano de avaliação de consulta que minimize o custo da avaliação da consulta; essa tarefa é chamada de *otimização da consulta*. O Capítulo 14 descreve a otimização da consulta com detalhes.

Quando o plano de consulta for escolhido, a consulta será avaliada com esse plano, e o resultado da consulta será gerado.

A seqüência de etapas já descrita para o processamento de uma consulta é representativa; nem todos os bancos de dados seguem exatamente essas etapas. Por exemplo, em vez de usar a representação da álgebra relacional, vários bancos de dados utilizam uma representação de árvore de análise anotada com base na estrutura da consulta SQL indicada. Porém, os conceitos que descrevemos aqui formam a base do processamento da consulta nos bancos de dados.

Para otimizar uma consulta, um otimizador de consulta precisa saber o custo de cada operação. Embora o custo exato seja difícil de calcular, pois depende de muitos parâmetros, como a memória real disponível à operação, é possível chegar a uma estimativa aproximada do custo de execução para cada operação.

Neste capítulo, estudamos como avaliar as operações individuais em um plano de consulta e como estimar seu custo; retornaremos à otimização da consulta no Capítulo 14. A próxima seção esboça como medir o custo de uma consulta. As seções "Operação de seleção" a "Outras operações" abordam a avaliação das operações da álgebra relacional. Várias operações podem ser agrupadas em uma **canalização** (ou **pipeline**), em que cada uma das operações começa atuando sobre suas tuplas de entrada, mesmo que elas estejam sendo geradas por outra operação. Na seção "Avaliação de expressões", examinamos como coordenar a execução de múltiplas operações em um plano de avaliação de consulta, em particular, como usar operações canalizadas para evitar a gravação de resultados intermediários em disco.

1. Para views materializadas, a expressão que define a view já foi avaliada e armazenada. Portanto, a relação armazenada pode ser utilizada no lugar da view, sendo substituída pela expressão que a define. As views recursivas são tratadas de forma diferente, por meio de um procedimento de ponto fixo, conforme discutimos nas seções "Consultas recursivas" do Capítulo 4 e "Recursão na Datalog" do Capítulo 5.

Medidas de custo da consulta

O custo de avaliação da consulta pode ser medido em termos de vários recursos diferentes, incluindo acessos ao disco, tempo de CPU para executar uma consulta e, em um sistema de banco de dados distribuído ou paralelo, o custo da comunicação (que discutiremos mais adiante, nos Capítulos 21 e 22). O tempo de resposta para um plano de avaliação de consulta (ou seja, a quantidade de tempo necessária para executar o plano), supondo que nenhuma outra atividade esteja ocorrendo no computador, responderia por todos esses custos e poderia ser usado como uma boa medida do custo do plano.

Porém, em grandes sistemas de banco de dados, o custo para acessar dados a partir do disco normalmente é o mais importante, pois os acessos ao disco são lentos em comparação com as operações na memória. Além do mais, as velocidades de CPU melhoraram muito mais rapidamente do que as velocidades de disco. Assim, é provável que o tempo gasto na atividade do disco continue a dominar o tempo total para executar uma consulta. O tempo de CPU tomado para uma tarefa é mais difícil de estimar, pois depende de detalhes de baixo nível do código da execução. Embora os otimizadores de consulta da vida real levem em consideração os custos de CPU, para simplificar, vamos ignorar esses custos e usar apenas os custos do acesso ao disco para medir o custo de um plano de avaliação de consulta.

Usamos o *número de transferências de bloco* do disco e o *número de buscas do disco* para medir o custo do acesso aos dados do disco. Se o subsistema de disco levar uma média de t_T segundos para transferir um bloco de dados, e tiver um tempo médio de acesso ao bloco (tempo de busca do disco mais latência rotacional) de t_S segundos, então uma operação que transfere b blocos e realiza S buscas levaria $b*t_T + S*t_S$ segundos. Os valores de t_T e t_S precisam ser calibrados para o sistema de disco utilizado, mas os valores típicos para os discos de alto nível seriam t_S = 4 milissegundos e t_T = 0,1 milissegundo, considerando um tamanho de bloco de 4 kilobytes e uma taxa de transferência de 40 megabytes por segundo.[2]

Podemos refinar nossas estimativas ainda mais distinguindo as leituras de bloco das escritas de bloco, pois estas normalmente custam o dobro das leituras (isso porque os sistemas de disco lêem setores de volta depois que eles são escritos, para verificar se a escrita teve sucesso). Para simplificar, vamos ignorar esse detalhe e deixar para você o desenvolvimento de estimativas de custo mais exatas para as várias operações.

2. Alguns sistemas de banco de dados realizam buscas de teste e transferências de bloco para estimar os custos médios de busca e transferência de bloco, como parte do processo de instalação de software.

As estimativas de custo que oferecemos não incluem o custo da escrita do resultado final de uma operação de volta ao disco. Essas são levadas em consideração separadamente onde for exigido. Os custos de todos os algoritmos que consideramos dependem do tamanho do buffer na memória principal. No melhor dos casos, todos os dados podem ser lidos para os buffers, e o disco não precisa ser acessado novamente. No pior dos casos, consideramos que o buffer pode manter apenas alguns blocos de dados – aproximadamente um bloco por relação. Ao representar estimativas de custo, geralmente consideramos o pior caso.

Além disso, embora consideremos que os dados precisam ser lidos do disco inicialmente, é possível que um bloco acessado já esteja presente no buffer da memória. Novamente, por simplicidade, ignoramos esse efeito; como resultado, o custo real de acesso ao disco durante a execução de um plano pode ser menor que o custo estimado.

Operação de seleção

No processamento da consulta, a **varredura do arquivo** é o operador de menor nível para acessar dados. As varreduras de arquivo são algoritmos de busca que localizam e apanham registros que cumprem uma condição de seleção. Nos sistemas relacionais, uma varredura de arquivo permite que uma relação inteira seja lida nos casos em que a relação é armazenada em um único arquivo dedicado.

Algoritmos básicos

Considere uma operação de seleção sobre uma relação cujas tuplas são armazenadas juntas em um arquivo. Dois algoritmos de varredura para implementar a operação de seleção são:

- A1 (**busca linear**). Em uma busca linear, o sistema varre cada bloco do arquivo e testa todos os registros para ver se satisfazem a condição de seleção. Uma busca inicial é necessária para acessar o primeiro bloco do arquivo. Caso os blocos do arquivo não estejam armazenados de forma adjacente, podem ser necessárias buscas extras, mas vamos ignorar esse efeito para simplificar.

  O custo da busca linear, em termos do número de operações de disco, é uma busca mais br transferências de bloco no arquivo ou, de forma equivalente, o custo de tempo é $t_S + b_r * t_T$.

  Para uma seleção sobre um atributo de chave, o sistema pode terminar a varredura se o registro exigido for encontrado, sem examinar os outros registros da relação. As seleções sobre atributos de chave possuem um custo de transferência médio de $b_r/2$, mas ainda têm um custo do pior caso de b_r transferências de bloco, além de uma busca.

Embora possa ser mais lento do que outros algoritmos para implementar a seleção, o algoritmo de busca linear pode ser aplicado a qualquer arquivo, independente da classificação do arquivo ou da disponibilidade de índices, ou da natureza da operação de seleção. Os outros algoritmos que estudaremos não se aplicam a todos os casos, mas, quando se aplicam, eles geralmente são mais rápidos do que a busca linear.

- **A2 (busca binária).** Se o arquivo for ordenado por um atributo, e a condição de seleção for uma comparação de igualdade sobre o atributo, podemos usar uma busca binária para localizar os registros que satisfazem a seleção. O sistema realiza a busca binária sobre os blocos do arquivo.

  No pior caso, o número de blocos que precisam ser examinados para encontrar um bloco contendo os registros exigidos é $\lceil \log_2(b_r) \rceil$, onde b_r indica o número de blocos no arquivo. Cada um desses acessos de bloco exige uma busca de disco, além de uma transferência de bloco, e o custo do tempo, portanto, é $\lceil \log_2(b_r) \rceil * (t_T + t_S)$.

  Se a seleção for sobre um atributo não-chave, mais de um bloco poderá conter os registros solicitados, e o custo da leitura de blocos extras precisa ser somado à estimativa de custo. Podemos estimar esse número estimando o tamanho do resultado da seleção (que veremos na seção "Estimando estatísticas de resultados de expressão" do Capítulo 14), e dividindo-o pelo número médio de registros que são armazenados por bloco da relação. Esses blocos são considerados como armazenados de forma contígua, de modo que só pagamos um custo de transferência t_T por bloco extra.

Seleções usando índices

As estruturas de índice são referenciadas como **caminhos de acesso**, pois oferecem um caminho pelo qual os dados podem ser localizados e acessados. No Capítulo 12, dissemos que é eficiente ler os registros de um arquivo em uma ordem que corresponde de perto à ordem física. Lembre-se de que um *índice primário* (também chamado de *índice agrupado*) é um índice que permite que os registros de um arquivo sejam lidos em uma ordem que corresponde à ordem física no arquivo. Um índice que não é o índice primário é chamado de *índice secundário*.

Os algoritmos de busca que usam um índice são considerados **varreduras de índice**. Os índices ordenados, como as árvores B$^+$, também permitem o acesso às tuplas em uma ordem classificada, que é útil para implementar consultas de intervalo. Embora os índices possam oferecer acesso rápido, direto e ordenado, eles impõem a sobrecarga de acesso aos blocos que contêm o índice. Usamos o predicado de seleção para nos guiar na escolha do índice a usar no processamento da consulta. Os algoritmos de busca que usam um índice são:

- **A3 (índice primário, igualdade sobre chave).** Para uma comparação de igualdade sobre um atributo de chave com um índice primário, podemos usar o índice para apanhar um único registro que satisfaz a condição de igualmente correspondente.

  Se uma árvore B$^+$ for usada, o custo da operação, em termos de operações de E/S, é igual à altura da árvore mais uma E/S para apanhar o registro;[3] cada uma dessas operações de E/S exige uma busca e uma transferência de bloco. Assim, o custo é $(h_i+1)*(t_T+t_S)$, onde h_i indica a altura do índice.[4]
- **A4 (índice primário, igualdade sobre não chave).** Podemos apanhar vários registros usando um índice primário quando a condição de seleção especifica uma comparação de igualmente sobre um atributo não de chave, *A*. A única diferença do caso anterior é que vários registros podem ter de ser apanhados. Porém, os registros seriam armazenados consecutivamente no arquivo, pois o arquivo é classificado por chave de busca.

  O custo da operação depende da altura da árvore, mais o número de blocos contendo registros com a chave de busca especificada. Uma busca é necessária para cada nível da árvore. Além disso, uma busca é necessária para chegar ao primeiro bloco contendo um registro desejado; os blocos restantes são armazenados consecutivamente e não exige mais buscas. Especificamente, o custo é $h_i * (t_T + t_S) + t_S + b^* t_T$, onde h_i é a altura da árvore e b é o número de blocos contendo registros com a chave de busca especificada.
- **A5 (índice secundário, igualdade).** As seleções especificando uma condição de igualdade podem usar um índice secundário. Essa estratégia pode apanhar um único registro se a condição de igualdade for sobre uma chave; vários registros podem ser apanhados se o campo de indexação não for uma chave.

  No primeiro caso, somente um registro é apanhado, o que exige uma operação de E/S para cada nível da árvore

3. Se uma organização de arquivo em árvore B$^+$ for utilizada, a E/S extra não é necessária, pois os registros estão armazenados no nível de folha da árvore. Ajustes semelhantes deverão ser feitos para alguns dos algoritmos descritos mais adiante nesta seção, se uma organização de arquivo de árvore B$^+$ for utilizada.

4. Os otimizadores da vida real normalmente consideram que a raiz da árvore estará presente no buffer na memória, por ser acessada com freqüência. Alguns otimizadores até mesmo consideram que tudo menos o nível de folha da árvore está presente na memória, pois são acessados de modo relativamente freqüente, e normalmente menos de 1% dos nós de uma árvore B$^+$ são nós não de folha. A fórmula do custo pode ser modificada de modo apropriado.

mais uma operação de E/S para apanhar o registro. Cada operação de E/S exige uma busca e uma transferência de bloco. O custo de tempo nesse caso é o mesmo daquele para um índice primário (caso A3).

No segundo caso, cada registro pode ser residente em um bloco diferente, o que pode resultar em uma operação de E/S por registro apanhado, com cada operação de E/S exigindo uma busca e uma transferência de bloco. O custo poderia se tornar ainda pior do que o da busca linear se uma grande quantidade de registros for apanhada. O custo de tempo, nesse caso, é de $(h_i+n) * (t_S + t_T)$, onde n é o número de registros apanhados.[5]

Conforme descrito na seção "Índices secundários e relocação de registros", quando os registros são armazenados em uma organização de arquivo de árvore B+ ou outras organizações de arquivo que possam exigir relocação de registros, os índices secundários normalmente não armazenam ponteiros para os registros.[6] Em vez disso, índices secundários armazenam os valores dos atributos usados como chave de busca em uma organização de arquivo de árvore B+. O acesso a um registro por meio desse índice secundário, portanto, é mais dispendioso: primeiro, o índice secundário é pesquisado para encontrar os valores de chave de busca do índice primário, depois o índice primário é pesquisado para encontrar os registros. A fórmula de custo descrita para os índices secundários precisa ser modificada corretamente se esses índices forem usados.

Seleções envolvendo comparações

Considere uma seleção da forma $\sigma_{A\leq v}(r)$. Podemos implementá-la usando uma busca linear ou binária, ou usando índices em uma das seguintes maneiras:

- **A6 (índice primário, comparação).** Um índice primário ordenado (por exemplo, um índice primário de árvore B+) pode ser usado quando a condição de seleção é uma comparação. Para condições de comparação na forma $A > v$ ou $A \geq v$, um índice primário sobre A pode ser usado para direcionar a recuperação de tuplas, da seguinte forma. Para $A \geq v$, pesquisamos o valor v no índice para encontrar a primeira tupla no arquivo que tenha um valor $A = v$. Uma varredura de arquivo começando por essa tupla até o final do arquivo retorna todas as tuplas que satisfazem a condição. Para $A > v$, a varredura de arquivo começa com a primeira tupla de modo que $A > v$. A estimativa de custo para esse caso é idêntica à usada para o caso A4.

  Para as comparações na forma $A < v$ ou $A \leq v$, uma pesquisa de índice não é necessária. Para $A < v$, usamos uma varredura de arquivo simples, começando com o início do arquivo e continuando até (mas sem incluir) a primeira tupla com atributo $A = v$. O caso de $A \leq v$ é semelhante, exceto que a varredura continua até (mas sem incluir) a primeira tupla com atributo $A > v$. De qualquer forma, o índice não é útil.
- **A7 (índice secundário, comparação).** Podemos usar um índice ordenado secundário para guiar a recuperação para as condições de comparação envolvendo $<$, $\leq$, $\geq$ ou $>$. Os blocos de índice de menor nível são varridos, seja do menor valor até v (para $< e \leq$) ou de v até o valor máximo (para $> e \geq$).

  O índice secundário oferece ponteiros aos registros, mas, para chegar aos registros reais, temos de apanhar os registros usando os ponteiros. Essa etapa pode exigir uma operação de E/S para cada registro apanhado, pois os registros consecutivos podem estar em diferentes blocos de disco; como antes, cada operação de E/S exige uma busca de disco e uma transferência de bloco. Se o número de registros apanhados for grande, o uso do índice secundário pode ser ainda mais dispendioso do que o uso da busca linear. Portanto, o índice secundário só deve ser usado se muito poucos registros forem selecionados.

Implementação de seleções complexas

Até aqui, consideramos apenas condições de seleção simples, na forma $A\ op\ B$, onde op é uma operação de igualdade ou comparação. Agora, vamos considerar predicados de seleção mais complexos.

- **Conjunção:** uma *seleção conjuntiva* é uma seleção na forma

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$$

- **Disjunção:** uma *seleção disjuntiva* é uma seleção na forma

$$\sigma_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n}(r)$$

Uma condição disjuntiva é satisfeita pela união de todos os registros que satisfazem as condições individuais, simples θ_i.

5. Se o buffer na memória for grande, o bloco contendo o registro já poderá estar no buffer. É possível construir uma estimativa do custo *médio* ou *esperado* da seleção, levando em conta a probabilidade de o bloco que contém o registro já estar no buffer. Para buffers grandes, essa estimativa se tornará muito menor do que a estimativa do pior caso.
6. Lembre-se de que, se as organizações de arquivo de árvore B+ forem usadas para armazenar relações, os registros podem ser movidos entre os blocos quando os nós de folha forem divididos ou mesclados, e quando os registros forem distribuídos.

- **Negação**: o resultado de uma seleção $\sigma_{-\theta}(r)$ é um conjunto de tuplas de r para as quais a condição θ é avaliada como falsa. Na ausência de nulos, esse conjunto é simplesmente o conjunto de tuplas que não estão em $\sigma_{\theta}(r)$.

Podemos implementar uma operação de seleção envolvendo uma conjunção ou uma disjunção de condições simples usando um dos algoritmos a seguir:

- **A8 (seleção conjuntiva usando um índice)**. Primeiro, determinamos se um caminho de acesso está disponível para um atributo em uma das condições simples. Se houve algum, um dos algoritmos de seleção de A2 a A7 poderá apanhar os registros que satisfazem essa condição. Completamos a operação testando, no buffer da memória, se cada registro satisfaz as condições simples restantes ou não.

  Para reduzir o custo, escolhemos um θ_i e um dos algoritmos de A1 a A7 para o qual a combinação resulta no menor custo para $\sigma_{\theta_i}(r)$. O custo do algoritmo A8 é dado pelo custo do algoritmo escolhido.
- **A9 (seleção conjuntiva usando índice composto)**. Um *índice composto* apropriado (ou seja, um índice sobre vários atributos) pode estar disponível para algumas seleções conjuntivas. Se a seleção especificar uma condição de igualdade sobre dois ou mais atributos, e um índice composto existir sobre esses campos de atributo combinados, então o índice pode ser pesquisado diretamente. O tipo de índice determina qual dos algoritmos A3, A4 ou A5 será utilizado.
- **A10 (seleção conjuntiva pela interseção de identificadores)**. Outra alternativa para implementar operações de seleção conjuntiva envolve o uso de ponteiros de registro ou identificadores de registro. Esse algoritmo exige índices com ponteiros de registro, sobre os campos envolvidos nas condições individuais. O algoritmo varre cada índice para ponteiros de tuplas que satisfazem uma condição individual. A interseção de todos os ponteiros apanhados é o conjunto de ponteiros para tuplas que satisfazem a condição conjuntiva. O algoritmo, então, usa os ponteiros para apanhar os registros reais. Se os índices não estiverem disponíveis sobre todas as condições individuais, então o algoritmo testa os registros apanhados em relação às condições restantes.

  O custo do algoritmo **A10** é a soma dos custos das varreduras de índice individuais, mais o custo da leitura dos registros na interseção das listas de ponteiros recuperadas. Esse custo pode ser reduzido pela classificação da lista de ponteiros e leitura dos registros na ordem classificada. Com isso, (1) todos os ponteiros para registros em um bloco vêm juntos, e portanto todos os registros selecionados no bloco podem ser apanhados por meio de uma única operação de E/S, e (2) os blocos são lidos em ordem classificada, reduzindo o movimento do braço do disco. A próxima seção descreve os algoritmos de classificação.
- **A11 (seleção disjuntiva pela união de identificadores)**. Se os caminhos de acesso estiverem disponíveis em todas as condições de uma seleção disjuntiva, cada índice é varrido para os ponteiros de tuplas que satisfazem a condição individual. A união de todos os ponteiros apanhados gera o conjunto de ponteiros para todas as tuplas que satisfazem a condição disjuntiva. Depois, usamos os ponteiros para apanhar os registros reais.

  Porém, se mesmo uma das condições não tiver um caminho de acesso, temos de realizar uma varredura linear da relação para encontrar tuplas que satisfazem a condição. Portanto, se houver até mesmo uma condição desse tipo na disjunção, o método de aceso mais eficiente será uma varredura linear, com a condição disjuntiva testada sobre cada tupla durante a varredura.

A implementação de seleções com condições de negação fica como um exercício para você (Exercício prático 13.6).

Classificação

A classificação de dados desempenha um papel importante nos sistemas de banco de dados por dois motivos. Primeiro, as consultas SQL podem especificar que a saída seja classificada. Segundo, e igualmente importante para o processamento da consulta, várias das operações relacionais, como junções, podem ser implementadas de forma eficiente se as relações de entrada forem classificadas primeiro. Assim, discutimos a classificação aqui, antes de discutirmos a operação de junção na próxima seção.

Podemos classificar uma relação criando um índice sobre a chave de classificação, e depois usado esse índice para ler a relação em ordem classificada. Porém, esse processo ordena a relação apenas *logicamente*, por intermédio de um índice, e não *fisicamente*. Logo, a leitura de tuplas na ordem classificada pode levar a um acesso de disco (busca de disco mais transferência de bloco) para cada registro, o que pode ser muito dispendioso, pois a quantidade de registros pode ser muito maior do que o número de blocos. Por esse motivo, pode ser preferível ordenar os registros fisicamente.

O problema de classificação foi bastante estudado, tanto por relações que se encaixam inteiramente na memória principal quanto por relações que são maiores do que a memória. No primeiro caso, técnicas de classificação padrão, como quick-sort, podem ser utilizadas. Aqui, discutimos como lidar com o segundo caso.

A classificação de relações que não cabem na memória é chamada de **classificação externa**. A técnica mais utilizada

para a classificação externa é o **algoritmo sort-merge externo**. Descrevemos o algoritmo sort-merge externo em seguida. Considere que M indique o número de frames de página no buffer da memória principal (o número de blocos de disco cujo conteúdo pode ser mantido em buffer na memória principal).

1. No primeiro estágio, diversas **rodadas** classificadas são criadas; cada rodada é classificada, mas contém apenas alguns dos registros da relação.

   ```
   i = 0;
   repeat
     lê M blocos da relação, ou o restante da relação,
       o que for menor;
     classifica a parte da relação na memória;
     grava dados classificados no arquivo da rodada R_i;
   i = i + 1;
   until o final da relação
   ```

2. No segundo estágio, as rodadas são *mescladas*. Suponha, por enquanto, que o número total de rodadas, N, seja menor que M, de modo que podemos alocar um frame de página para cada rodada e ficar com espaço para manter uma página de saída. O estágio merge opera da seguinte forma:

   ```
   lê um bloco de cada um dos N arquivos R_i para
   uma página de buffer na memória;
   repeat
     escolhe a primeira tupla (na ordem de
   classificação) entre todas as páginas de buffer;
     escreve a tupla na saída e a excluir da página de
   buffer;
     if a página de buffer de qualquer rodada R_i for
   vazia and not end-of-file(R_i)
       then lê o próximo bloco de R_i para a página de
   buffer;
   until todas as páginas de buffer estarem vazias
   ```

A saída do estágio de merge é a relação classificada. O arquivo de saída é colocado em buffer para reduzir o número de operações de gravação em disco. A operação de merge anterior é uma generalização do merge bidirecional utilizado pelo algoritmo de sort-merge padrão na memória; ele mescla N rodadas, de modo que é chamado **merge de N vias**.

Em geral, se a relação for muito maior do que a memória, pode haver M ou mais rodadas geradas no primeiro estágio, e não é possível alocar um frame de página para cada rodada durante o estágio de merge. Nesse caso, a operação de merge prossegue em vários passos. Como existe memória suficiente para M–1 páginas de buffer de entrada, cada merge pode usar M–1 rodadas como entrada.

A *passada* inicial funciona desta maneira: ela mescla as primeiras M–1 rodadas (conforme descrito no item 2, anteriormente) para chegar a uma única rodada para a próxima passada. Depois, ela mescla as próximas M–1 rodadas de modo semelhante, e assim por diante, até que tenha processado todas as rodadas iniciais. Neste ponto, o número de rodadas já foi reduzido por um fator de M–1. Se esse número reduzido de rodadas ainda for maior ou igual a M, outra passada é feita, com as rodadas criadas pela primeira passada servindo de entrada. Cada passada reduz o número de rodadas por um fator de M–1. As passadas são repetidas tantas vezes quantas forem necessárias, até que o número de rodadas seja menor que M; uma passada final, então, gera a saída classificada.

A Figura 13.3 ilustra as etapas do sort-merge externo para uma relação de exemplo. Para fins de ilustração, consideramos que somente uma tupla se encaixa em um bloco ($f_r = 1$), e consideramos que a memória mantém no máximo três frames de página. Durante o estágio de merge, dois frames de página são usados para entrada e um para saída.

Calculamos o custo do acesso ao disco para o sort-merge externo desta maneira: considere que b_r indica o número de blocos contendo registros da relação r. O primeiro estágio lê cada bloco da relação e os grava novamente, dando um total de $2b_r$ transferências de bloco. O número mínimo de rodadas é $\lceil b_r/M \rceil$. Como o número de rodadas diminui por um fator de M–1 em cada passada de merge, o número total de passadas de merge exigido é $\lceil \log_{M-1}(b_r/M) \rceil$. Cada uma dessas passadas lê cada bloco da relação uma vez e a grava uma vez, com duas exceções. Primeiro, a passada final pode produzir a saída classificada sem gravar seu resultado em disco. Em segundo lugar, pode haver rodadas que não são lidas ou gravadas durante uma passada – por exemplo, se houver M rodadas a serem mescladas em uma passada, M–1 são lidas e mescladas, e uma rodada não é acessada durante a passada. Ignorando as economias (relativamente pequenas) devidas a esse último efeito, o número total de transferências de bloco para a classificação externa da relação é

$$b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$$

Aplicando essa equação ao exemplo da Figura 13.3, obtemos um total de 12*(4 + 1) = 60 transferências de bloco, como você pode verificar pela figura. Observe que esses números não incluem o custo da escrita do resultado final.

Também precisamos somar os custos de busca de disco. A geração da rodada exige buscas para a leitura de dados em cada uma das duas rodadas, além da gravação da rodada. Durante a fase de merge, se os dados lerem b_b blocos ao mesmo tempo em cada rodada (ou seja, b_b blocos de buffer são alocados a cada rodada), então cada passada de merge

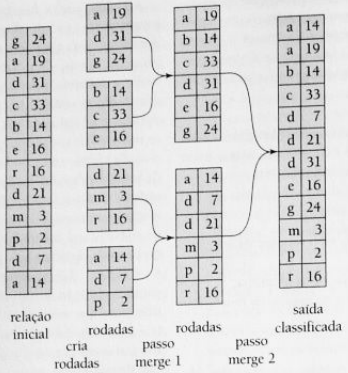


Figura 13.3 Classificação externa usando sort-merge.

exigiria em torno de $\lceil b_r/b_b \rceil$ buscas para leitura de dados.[7] Embora a saída seja gravada sequencialmente, se ela estiver no mesmo disco das rodadas de entrada, a cabeça de leitura pode ter se movido entre as gravações de blocos consecutivos. Assim, teríamos de incluir um total de $2\lceil b_r/b_b \rceil$ buscas para cada passada, exceto pela passada final (pois assumimos que o resultado final não está gravado de volta ao disco). O número total de buscas é, portanto,

$$2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2\lceil \log_{M-1}(b_r/M) \rceil - 1)$$

Aplicando essa equação ao exemplo da Figura 13.3, obtemos um total de $8 + 12*(2*2 - 1) = 44$ buscas de disco se definirmos o número de blocos de buffer por rodada, b_b, como 1.

Operação de junção

Nesta seção, estudamos vários algoritmos para calcular a junção de relações, e analisamos seus respectivos custos.

Usamos o termo **equi-junção** para nos referir a uma junção da forma $r \bowtie_{r.A=s.B} s$, onde A e B são atributos ou conjuntos de atributos das relações r e s, respectivamente.

Usamos como exemplo corrente a expressão

$$depositante \bowtie cliente$$

usando os mesmos esquemas de relação que usamos nos Capítulos 2 e 3. Consideramos a seguinte informação sobre as duas relações:

- Número de registros de *cliente*: $n_{cliente} = 10.000$.
- Número de blocos de cliente: $b_{cliente} = 400$.
- Número de registros de *depositante*: $n_{depositante} = 5000$.
- Número de blocos de *depositante*: $b_{depositante} = 100$.

Junção de loop aninhado

A Figura 13.4 é um algoritmo simples para calcular a junção, $r \bowtie_\theta s$, de duas relações r e s. Esse algoritmo é denominado algoritmo de junção **de loop aninhado**, pois basicamente consiste em um par de loops **for** aninhados. A relação r é denominada **relação externa** e a relação s é a **relação interna** da junção, pois o loop para r delimita o loop para s. O algoritmo usa a notação $t_r \cdot t_s$, onde t_r e t_s são tuplas; $t_r \cdot t_s$ indica a tupla construída pela concatenação dos valores de atributo das tuplas t_r e t_s.

Assim como o algoritmo de varredura de arquivo linear para seleção, o algoritmo de junção de loop aninhado não exige índices, e pode ser usado independente da condição de junção. A extensão do algoritmo para calcular a junção natural é direta, pois a junção natural pode ser expressa como uma junção theta seguida pela eliminação dos atributos repetidos por uma projeção. A única mudança exigida é uma etapa extra de exclusão de atributos repetidos da tupla $t_r \cdot t_s$, antes de somá-la ao resultado.

7. Para ser mais exato, como lemos cada rodada separadamente e podemos obter menos do que b_b blocos ao ler o final de uma rodada, podemos exigir uma busca extra para cada rodada. Ignoramos esse detalhe para simplificar.

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        par de testes (t_r,t_s) para ver se satisfazem a condição de junção θ
        nesse caso, soma t_r · t_s ao resultado.
    end
end
```

Figura 13.4 Junção de loop aninhado.

O algoritmo de junção de loop aninhado é dispendioso, pois examina cada par de tuplas nas duas relações. Considere o custo do algoritmo de junção de loop aninhado. O número de pares de tuplas a serem consideradas é $n_r * n_s$, onde n_r indica o número de tuplas em r, e n_s indica o número de tuplas em s. Para cada registro em r, temos de realizar uma varredura completa sobre s. No pior caso, o buffer pode manter apenas um bloco de cada relação, e um total de $n_r * b_s + b_r$ transferências de bloco seriam necessárias, onde b_r e b_s indicam o número de blocos contendo tuplas de r e s, respectivamente. Só precisamos de uma busca para cada varredura na relação interna s, pois ela é lida seqüencialmente, e de b_r buscas para ler r, levando a um total de $n_r + b_r$ buscas. No melhor caso, existe espaço suficiente para as duas relações caberem simultaneamente na memória, de modo que cada bloco teria de ser lido apenas uma vez; logo, somente $b_r + b_s$ transferências de bloco seriam necessárias, junto com 2 buscas.

Se uma das relações couber inteiramente na memória principal, será benéfico usar essa relação como a relação interna, pois a relação interna seria lida apenas uma vez. Portanto, se s for pequeno o suficiente para caber na memória principal, nossa estratégia exige apenas um total de $b_r + b_s$ transferências de bloco e 2 buscas – o mesmo custo do caso em que as duas relações cabem na memória.

Agora considere a junção natural de *depositante* e *cliente*. Suponha, por enquanto, que não tenhamos índices em qualquer relação, e que não queiramos criar qualquer índice. Podemos usar loops aninhados para calcular a junção; suponha que o *depositante* seja a relação externa e *cliente* seja a relação interna na junção. Teremos de examinar 5000 * 10.000 = 50 * 10^6 pares de tuplas. No pior caso, o número de transferências de bloco é 5000 * 400 + 100 = 2.000.100, mais 5000 + 100 = 5100. Porém, no cenário do melhor caso, podemos ler ambas as relações apenas uma vez, e realizar a computação. Essa computação requer no máximo 100 + 400 = 500 transferências de bloco, mais 2 buscas – uma melhoria significativa em relação ao cenário do pior caso. Se tivéssemos usado *cliente* como relação para o loop externo e *depositante* para o loop interno, o custo do pior caso da nossa estratégia final teria sido 10.000 * 100 + 400 = 1.000.400 transferências de bloco, mais 10.400 buscas de disco. O número de transferências de disco é muito menor, e embora o número de buscas seja maior, o custo geral é reduzido, considerando $t_s = 4$ milissegundos e $t_T = 0{,}1$ milissegundo.

Junção de loop aninhado em bloco

Se o buffer for muito pequeno para manter qualquer relação inteiramente na memória, ainda podemos obter uma economia importante nos acessos de bloco se processarmos as relações com base em cada bloco, em vez de tupla. A Figura 13.5 mostra a **junção de loop aninhado em bloco**, que

```
for each bloco B_r of r do begin
    for each bloco B_s of s do begin
        for each tupla t_r in B_r do begin
            for each tupla t_s in B_s do begin
                testa par (t_r,t_s) para ver se satisfaz a condição de junção
                se satisfizer, acrescente t_r · t_s ao resultado.
            end
        end
    end
end
```

Figura 13.5 Junção de loop aninhado em bloco.

é uma variante da junção de loop aninhado, em que cada bloco da relação interna é emparelhado com cada bloco na relação externa. Dentro de cada par de blocos, cada tupla em um bloco é emparelhada com cada tupla no outro bloco, para gerar todos os pares de tuplas. Como antes, todos os pares de tuplas que satisfazem a condição da junção são acrescentados ao resultado.

A principal diferença no custo entre a junção de loop aninhado em bloco e a junção de loop aninhado básico é que, no pior caso, cada bloco na relação interna s é lido apenas uma vez para cada *bloco* na relação externa, em vez de uma vez para cada *tupla* na relação externa. Assim, no pior caso, haverá um total de $b_r * b_s + b_r$ transferências de bloco, onde b_r e b_s indicam o número de blocos contendo registros de r e s, respectivamente. Cada varredura da relação interna exige uma busca, e a varredura da relação externa exige uma busca por bloco, levando a um total de $2 * b_r$ buscas. Por certo, é mais eficiente usar a relação menor como relação externa, caso nenhuma das relações caiba na memória. No melhor caso, em que a relação interna cabe na memória, haverá $b_r + b_s$ transferências de bloco e apenas 2 buscas (escolheríamos a menor relação como a relação interna neste caso).

Agora, retorne ao nosso exemplo de computação de *depositante* ⋈ *cliente*, usando o algoritmo de junção de loop aninhado em bloco. No pior caso, temos de ler cada bloco de *cliente* uma vez para cada bloco de *depositante*. Assim, no pior caso, um total de 100 * 400 + 100 = 40.100 transferências de bloco mais 2 * 100 = 200 buscas são necessárias. O custo é uma melhoria significativa em relação às 5000 * 400 + 100 = 2.000.100 transferências de bloco mais 5100 buscas necessárias no pior caso para a junção de loop aninhado básica. O custo no melhor caso continua sendo o mesmo – a saber, 100 + 400 = 500 transferências de bloco e 2 buscas.

O desempenho dos procedimentos de loop aninhado e loop aninhado em bloco pode ser melhorado ainda mais:

- Se os atributos de união em uma junção natural ou em uma equijunção formarem uma chave sobre a relação interna, então, para cada tupla de relação externa, o loop interno pode terminar assim que a primeira combinação for encontrada.
- No algoritmo de loop aninhado em bloco, em vez de usar blocos de disco como unidade de bloco para a relação externa, podemos usar o maior tamanho que pode caber na memória, enquanto deixamos espaço suficiente para os buffers na relação interna e da saída. Em outras palavras, se a memória tiver M blocos, lemos $M - 2$ blocos da relação externa de cada vez, e quando lemos cada bloco da relação interna, nós o unimos a todos os $M - 2$ blocos da relação externa. Essa mudança reduz o número de varreduras da relação interna de b_r para $\lceil b_r / (M - 2) \rceil$, onde b_r é o número de blocos da relação externa. O custo total é, então, $\lceil b_r / (M - 2) \rceil * b_s + b_r$ transferências de bloco e $2 \lceil b_r / (M - 2) \rceil$ buscas.
- Podemos varrer o loop interno alternadamente, para a frente e para trás. Esse método de varredura ordena as solicitações para blocos de disco, de modo que os dados permanecendo no buffer a partir da varredura anterior podem ser reutilizados, reduzindo assim a quantidade necessária de acessos ao disco.
- Se um índice estiver disponível no atributo de junção do loop interno, podemos substituir as varreduras de arquivo por pesquisas de índice mais eficientes. A seção a seguir descreve essa otimização.

Junção de loop aninhado indexado

Em uma junção de loop aninhado (Figura 13.4), se um índice estiver disponível sobre o atributo de junção do loop interno, as pesquisas de índice podem substituir varreduras de arquivo. Para cada tupla t_r na relação externa r, o índice é usado para pesquisar tuplas em s que satisfarão a condição de junção com a tupla t_r.

Esse método de junção é denominado **junção de loop aninhado indexado**; ele pode ser usado com os índices existentes, e também com índices temporários criados para a única finalidade de avaliar a união.

Examinar as tuplas em s que satisfarão as condições de junção com determinada tupla t_r é basicamente uma seleção sobre s. Por exemplo, considere *depositante* ⋈cliente. Suponha que tenhamos uma tupla de *depositante* com *nome_cliente* "John". Então, as tuplas relevantes em s são aquelas que satisfazem a seleção $\sigma_{nome\_cliente = \text{John}}$".

O custo de uma junção de loop aninhado indexado pode ser calculado da seguinte maneira. Para cada tupla na relação externa r, uma pesquisa é realizada sobre o índice para s, e as tuplas relevantes são apanhadas. No pior caso, existe espaço no buffer para apenas uma página de r e uma página do índice. Então, as operações de E/S em b_r são necessárias para ler a relação r, onde b_r indica o número de blocos contendo registros de r; cada E/S exige uma busca e uma transferência de bloco, pois a cabeça de disco pode ter sido movida entre cada E/S. Para cada tupla em r, realizamos uma pesquisa de índice sobre s. Depois, o custo do tempo da junção pode ser calculado como $b_r * (t_T + t_S) + n_r * c$, onde n_r é o número de registros na relação r, e c é o custo de uma única seleção sobre s usando a condição de junção. Vimos, na seção "Operação de seleção", como estimar o custo de um único algoritmo de seleção (possivelmente usando índices); essa estimativa nos dá o valor de c.

A fórmula de custo indica que, se os índices estiverem disponíveis nas relações r e s, geralmente é mais eficiente usar aquele com menos tuplas como relação externa.

Por exemplo, considere uma junção de loop aninhado indexado de *depositante**join cliente*, com *depositante* como relação externa. Suponha também que *cliente* tenha um índice de árvore B+ primário sobre o atributo de junção *nome_cliente*, que contém 20 entradas na média em cada nó de índice. Como *cliente* tem 10.000 tuplas, a altura da árvore é 4, e mais um acesso é necessário para localizar os dados reais. Como $n_{depositante}$ é 5000, o custo total é 100 + 5000 * 5 = 25.100 acessos ao disco, cada um exigindo uma busca e uma transferência de arquivo. Ao contrário, como vimos antes, 40.100 transferências de bloco mais 200 buscas eram necessárias para uma junção de loop aninhado em bloco. Embora o número de transferências de bloco tenha diminuído, o custo da busca realmente aumentou, assim como o custo total, pois uma busca é consideravelmente mais dispendiosa do que uma transferência de bloco. Porém, se tivéssemos uma seleção sobre a relação *depositante* que reduzisse o número de linhas significativamente, a junção de loop aninhado indexado poderia ser muito mais rápida do que a junção de loop aninhado em bloco.

Junção merge

O algoritmo de **junção merge** (também chamado algoritmo **junção sort-merge**) pode ser usado para calcular as junções naturais e equijunções. Considere $r(R)$ e $s(S)$ como as relações cuja junção natural deve ser calculada, e considere $R \cap S$ como seus atributos comuns. Suponha que as duas relações sejam classificadas sobre os atributos $R \cap S$. Então, sua junção pode ser calculada por um processo muito semelhante ao estágio de merge no algoritmo merge-sort.

A Figura 13.6 mostra o algoritmo de junção merge. No algoritmo, *JoinAttrs* refere-se aos atributos em $R \cap S$, e $t_r \bowtie t_s$, onde t_r e t_s são tuplas que têm os mesmos valores

```
pr := endereço da primeira tupla de r;
ps := endereço da primeira tupla de s;
while (ps ≠ null and pr ≠ null) do
    begin
        t_s := tupla à qual ps aponta;
        S_s := {t_s};
        defina ps para apontar para próxima tupla de s;
        done := false;
        while (not done and ps ≠ null) do
            begin
            t_s' := tupla à qual ps aponta;
            if (t_s'[JoinAttrs] = t_s[JoinAttrs])
            then begin
                S_s := S_s ∪ {t_s'};
                defina ps para apontar para próxima tupla de s;
                end
            else done := true;
        end
        t_r := tupla à qual pr aponta;
        while (pr ≠ null and t_r[JoinAttrs] < t_s[JoinAttrs]) do
            begin
            defina pr para apontar para próxima tupla de r;
            t_r := tupla à qual pr aponta;
        end
    while (pr ≠ null and t_r[JoinAttrs] = t_s[JoinAttrs]) do
        begin
            for each t_s in S_s do
                begin
                    acrescenta t_s ⋈ t_r ao resultado;
            end
        defina pr para apontar para a próxima tupla de r;
        t_r := tupla à qual pr aponta;
    end
end.
```

Figura 13.6 Junção merge.

para *JoinAttrs*, indica a concatenação dos atributos das tuplas, seguido pela projeção de atributos repetidos. O algoritmo de junção merge associa um ponteiro a cada relação. Esses ponteiros apontam inicialmente para a primeira tupla das respectivas relações. Com o prosseguimento do algoritmo, os ponteiros se movem pela relação. Um grupo de tuplas de uma relação com o mesmo valor nos atributos de junção é lido para S_r. O algoritmo na Figura 13.6 *requer* que cada conjunto de tuplas S_x caiba na memória principal; mais adiante, nesta seção, veremos as extensões do algoritmo para evitar esse requisito. Depois, as tuplas correspondentes (se houver) da outra relação são lidas e processadas à medida que são lidas.

A Figura 13.7 mostra duas relações que são classificadas em seu atributo de junção *a*1. É instrutivo passar pelas etapas do algoritmo de junção merge sobre as relações mostradas na figura.

Como as relações estão em ordem classificada, as tuplas com o mesmo valor nos atributos de junção estão em ordem consecutiva. Por isso, cada tupla na ordem classificada precisa ser lida apenas uma vez e, como resultado, cada bloco também é lido apenas uma vez. Como é preciso apenas uma única passada por ambos os arquivos, o método de junção merge é eficiente; o número de transferências de bloco é igual à soma do número de blocos nos dois arquivos, $b_r + b_s$.

Supondo que b_b blocos de buffer sejam alocados a cada relação, o número de buscas de disco exigidas seria $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$ buscas de disco. Como as buscas são muito mais dispendiosas do que a transferência de dados, faz sentido alocar vários blocos de buffer a cada relação, desde que a memória extra esteja disponível. Por exemplo, com $t_T = 0.1$ milissegundo por bloco de 4 kilobytes, e $t_s = 4$ milissegundos, o tamanho do buffer é de 400 blocos (ou 1,6 megabyte), o tempo de busca seria 4 milissegundos para cada 40 milissegundos de tempo de transferência; em outras palavras, o tempo de busca seria apenas 10% do tempo de transferência.

Se qualquer uma das relações de entrada *r* e *s* não estiver classificada pelos atributos de junção, elas poderão ser classificadas primeiro, e depois o algoritmo de junção merge poderá ser usado. O algoritmo de junção merge também pode ser facilmente estendido a partir das junções naturais para o caso mais genérico das equijunções.

Como já dissemos, o algoritmo de junção merge da Figura 13.6 exige que o conjunto S_x de todas as tuplas com o mesmo valor para os atributos de junção caiba na memória principal. Esse requisito normalmente pode ser atendido, mesmo que a relação *s* seja grande. Se não puder ser atendido, uma junção do loop aninhado em bloco precisa ser realizado entre S_x e as tuplas em *r* com os mesmos valores para os atributos de junção. O custo geral da junção merge aumenta como resultado.

Também é possível realizar uma variação da operação junção merge sobre tuplas não classificadas, se existirem índices secundários sobre os dois atributos de junção. O algoritmo utiliza os índices para varrer os registros; como resultado, eles são apanhados em ordem classificada. Porém, essa variação apresenta uma desvantagem significativa, pois os registros podem estar espalhados pelos blocos do arquivo. Logo, o acesso a cada tupla poderia envolver o acesso a um bloco do disco, e isso é dispendioso.

Para evitar esse custo, podemos usar uma técnica de junção merge híbrida, que combina os índices com a junção merge. Suponha que uma das relações seja classificada; a outra não é, mas possui um índice de árvore B+ secundário sobre os atributos de junção. O **algoritmo de junção merge híbrida** mescla a relação classificada com as entradas de folha do índice de árvore B+ secundário. O arquivo de resultado contém tuplas da relação classificada e endereços para as tuplas da relação não classificada. O arquivo de resultado é, então, classificado sobre os endereços das tuplas da relação não classificada, permitindo a recuperação eficiente das tuplas correspondentes, na ordem de armazenamento física, para completar a junção. As extensões da técnica para li-

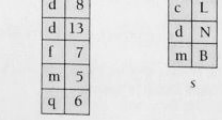


Figura 13.7 Relações classificadas para a junção merge.

dar com duas relações não classificadas são deixadas como um exercício para você.

Suponha que o esquema de junção merge seja aplicado ao nosso exemplo de *depositante* ⋈ cliente. O atributo de junção aqui é *nome_cliente*. Suponha que as relações já estejam classificadas sobre o atributo de junção *nome_cliente*. Nesse caso, a junção merge usa um total de 400 + 100 = 500 transferências de bloco. Se considerarmos que, no pior caso, somente um bloco de buffer é alocado para cada relação de entrada (ou seja, $b_b = 1$), um total de 400 + 100 = 500 buscas também seriam necessárias; na realidade, b_b pode ser definido como muito mais alto, pois só precisamos colocar blocos em buffer para duas relações, e o custo da busca seria muito menor.

Suponha que as relações não estejam classificadas, e o tamanho da memória seja o pior caso, somente três blocos. O custo é o seguinte:

1. Usando a fórmula que desenvolvemos na seção "Classificação", classificar a relação *cliente* exige $\lceil \log_{3-1}(400/3) \rceil = 8$ passadas de merge. A classificação da relação *cliente*, então, exige 400 * (2 $\lceil \log_{3-1}(400/3) \rceil + 1$), ou 6800 transferências de bloco, com mais 400 transferências para escrever o resultado. O número de buscas exigidas é 2 * $\lceil 400/3 \rceil$ + 400 *(2*8 – 1) ou 6268 buscas para classificação, e 400 buscas para escrever a saída, totalizando 6668 buscas, pois somente um bloco de buffer está disponível para cada rodada.
2. De modo semelhante, a classificação de *depositante* exige $\lceil \log_{3-1}(100/3) \rceil = 6$ passadas de merge e 100 * (2 $\lceil \log_{3-1}(100/3) \rceil + 1$), ou 1.300 transferências de bloco, com mais 100 transferências para escrever o resultado. O número de buscas exigidas para a classificação de *depositante* é 2 * $\lceil 100/3 \rceil$ + 100 *(2*6 – 1) = 1164, e 100 buscas são exigidas para escrever a saída, totalizando 1.264 buscas.
3. Finalmente, a mesclagem das duas relações exige 400 + 100 = 500 transferências de bloco e 500 buscas.

Assim, o custo total é 9.100 transferências de bloco mais 8.932 buscas se as relações não forem classificadas, e o tamanho da memória é de apenas 3 blocos.

Com um tamanho de memória de 25 blocos, e as relações não classificadas, o custo da classificação seguida por junção merge seria o seguinte:

1. A classificação da relação *cliente* pode ser feita com apenas uma etapa de mesclagem, e exige um total de apenas 400 * (2 $\lceil \log_{24}(400/25) \rceil$ +1) = 1200 transferências de bloco. De modo semelhante, a classificação de *depositante* exige 300 transferências de bloco. A escrita da saída classificada em disco exige 400 + 100 = 500 transferências de bloco, e a etapa de mesclagem exige 500 transferências de bloco para ler os dados de volta. Somando esses custos, temos um custo total de 2.500 transferências de bloco.
2. Se considerarmos que somente um bloco de buffer é alocado para cada rodada, o número de buscas exigidas nesse caso é 2 * $\lceil 400/25 \rceil$ + 400 + 400 = 832 buscas para classificar *cliente* e gravar a saída classificada em disco, e semelhantemente 2 * $\lceil 100/25 \rceil$ + 100 + 100 = 208 para *depositante*, mais 400 + 100 buscas para a leitura dos dados classificados na etapa de junção merge. Somando esses custos, temos um custo total de 1.640 buscas.

   O número de buscas pode ser reduzido bastante reservando-se mais blocos de buffer para cada rodada. Por exemplo, se 5 blocos de buffer forem alocados para cada rodada e para a saída a partir da mesclagem das 4 rodadas de *depositante*, o custo é reduzido para 2 * $\lceil 100/25 \rceil$ + $\lceil 100/5 \rceil$ + $\lceil 100/5 \rceil$ = 48 buscas, em vez de 208 buscas. Se a etapa de junção merge reservar 12 blocos cada para a colocação de *cliente* e *depositante* em buffer, o número de buscas para a etapa de junção merge cai para $\lceil 400/12 \rceil$ + $\lceil 100/12 \rceil$ = 43, em vez de 500. O número total de buscas é, portanto, 251.

Assim, o custo total é de 2.500 transferências de bloco mais 251 buscas se as relações não estiverem classificadas, e o tamanho da memória é de 25 blocos.

Junção hash

Assim como o algoritmo de junção merge, o algoritmo de junção hash pode ser usado para implementar junções naturais e equijunções. No algoritmo de junção hash, uma função de hash *h* é usada para particionar tuplas de duas relações. A idéia básica é particionar as tuplas de cada uma das relações em conjuntos que têm o mesmo valor de hash nos atributos de junção.

Consideramos que

- *h* é uma função de hash mapeando valores de *joinAttrs* para {0, 1, ..., n_h}, onde *joinAttrs* indica os atributos comuns de *r* e *s* usados na junção natural.
- $H_{r_0}, H_{r_1}, \ldots, H_{r_{n_h}}$ *indicam partições de r* tuplas, cada uma inicialmente vazia. Cada tupla $t_r \in r$ é colocada na partição H_{r_i}, *onde* $i = h(t_r[joinAttrs])$.
- $H_{s_0}, H_{s_1}, \ldots, H_{s_{n_h}}$ *indica partições de s* tuplas, cada uma inicialmente vazia. Cada tupla $t_s \in s$ é colocada na partição H_{s_i}, onde $i = h(t_s[joinAttrs])$.

A função de hash *h* deverá ter propriedades "virtuosas" de aleatoriedade e uniformidade, que discutimos no Capítulo 12. A Figura 13.8 representa o particionamento das relações.

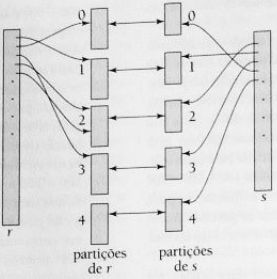


Figura 13.8 Particionamento de hash das relações.

Fundamentos

A idéia por trás do algoritmo de junção hash é esta: suponha que uma tupla *r* e uma tupla *s* satisfaçam a condição de junção; então, elas terão o mesmo valor para os atributos de junção. Se esse valor receber algum valor de hash *i*, a tupla *r* precisa estar em H_{ri} e a tupla *s* em H_{si}. Portanto, tuplas *r* em H_{ri} só precisam ser comparadas com tuplas *s* em H_{si}; elas não precisam ser comparadas com tuplas *s* em qualquer outra partição.

Por exemplo, se *d* for uma tupla em *depositante*, *c* uma tupla em *cliente* e *h* uma função de hash sobre os atributos *nome_cliente* das tuplas, então *d* e *c* só precisam ser testados se $h(c) = h(d)$. Se $h(c) \neq h(d)$ então *c* e *d* precisam ter diferentes valores para *nome_cliente*. Porém, se $h(c) = h(d)$, temos de testar *c* e *d* para ver se os valores em seus atributos de junção são iguais, pois é possível que *c* e *d* tenham diferentes valores de *nome_cliente* contendo o mesmo valor de hash.

A Figura 13.9 mostra os detalhes do algoritmo de junção **hash** para calcular a junção natural das relações *r* e *s*. Assim como no algoritmo de junção merge, $t_r \bowtie t_s$ indica a concatenação dos atributos das tuplas t_r e t_s, seguidos pela projeção de atributos repetidos. Após o particionamento das relações, o restante do código de junção hash realiza uma junção separada do loop aninhado indexado sobre cada um dos pares de partição *i*, para $i = 0, \ldots, n_h$. Para fazer isso, ele primeiro **monta** um índice de hash sobre cada H_{si} e depois

```
/* Partição s */
for each tupla t_s in s do begin
    i := h(ts[JoinAttrs]);
    H_si := H_si ∪ {t_s};
end
/* Partição r */
for each tupla t_r in r do begin
    i := h(t_r[JoinAttrs]);
    H_ri := H_ri ∪ {t_r};
end
/* Realiza junção sobre cada partição */
for i := 0 to n_h do begin
    lê H_si e monta um índice de hash na memória sobre ele
    for each tupla t_r in H_ri do begin
        sonda índice de hash em H_si para achar todas as tuplas t_s
            tais que t_s[JoinAttrs] = t_r[JoinAttrs]
        for each tupla t_s que combina in H_si do begin
            acrescenta t_r ⋈ t_s ao resultado
        end
    end
end
```

Figura 13.9 Junção hash.

sonda (ou seja, procura H_{s_i}) com as tuplas de H_{r_i}. A relação s é a **entrada de montagem**, e r é a **entrada de sonda**.

O índice de hash sobre H_{s_i} é montado na memória, de modo que não é preciso acessar o disco para apanhar as tuplas. A função de hash usada para montar esse índice de hash precisa ser diferente da função de hash *h* usada anteriormente, mas ainda é aplicada apenas aos atributos de junção. No curso da junção de loop aninhado indexado, o sistema usa esse índice de hash para apanhar registros que combinam com registros na entrada de sonda.

As fases de montagem e sonda exigem apenas uma única passada pelas entradas de montagem e sonda. É simples estender o algoritmo de junção hash para calcular equijunções gerais.

O valor n_h precisa ser escolhido para ser grande o suficiente, de modo que, para cada *i*, as tuplas na partição H_{s_i} da relação de montagem e o índice de hash sobre a partição caibam na memória. Não é preciso que as partições da relação de sonda caibam na memória. Claramente, é melhor usar a menor relação de entrada como relação de montagem. Se o tamanho da relação de montagem for de b_s blocos, então, para cada uma das partições n_h ter tamanho menor ou igual a *M*, n_h precisa ser pelo menos $\lceil b_s / M \rceil$. Mais precisamente, temos de levar em conta o espaço extra ocupado pelo índice de hash sobre a partição, de modo que n_h seja correspondentemente maior. Para simplificar, às vezes ignoramos o requisito do índice de hash em nossa análise.

Particionamento recursivo

Se o valor de n_h for maior ou igual ao número de frames de página da memória, as relações não podem ser particionadas em uma passada, pois não haverá páginas de buffer suficientes. Em vez disso, o particionamento precisa ser feito em passadas repetidas. Em uma passada, a entrada pode ser dividida em no máximo tantas partições quantos forem os frames de página que estiverem disponíveis para uso como buffers de saída. Cada bucket gerado por uma passada é lido separadamente e particionado novamente na próxima passada, para criar partições menores. A função de hash usada em uma passada, naturalmente, é diferente daquela usada na passada anterior. O sistema repete essa divisão da entrada até que cada partição da entrada de montagem caiba na memória. Esse particionamento é denominado **particionamento recursivo**.

Uma relação não precisa de particionamento recursivo se $M > n_h + 1$, ou, de forma equivalente, $M > (b_s / M) + 1$, o que pode ser simplificado (aproximadamente) para $M > \sqrt{b_s}$. Por exemplo, considere um tamanho de memória de 12 megabytes, divididos em blocos de 4 kilobytes; ela teria um total de 3K (3072) blocos. Podemos usar uma memória desse tamanho para particionar as relações de tamanho até 3K * 3K, que é 36 gigabytes. De modo semelhante, uma relação com tamanho de 1 gigabyte exige apenas um pouco mais de $\sqrt{256K}$ blocos, ou 2 megabytes, para evitar o particionamento recursivo.

Tratamento de estouros

O **estouro da tabela de hash** ocorre no particionamento *i* da relação de montagem s se o índice de hash sobre H_{s_i} for maior do que a memória principal. O estouro da tabela de hash pode ocorrer se houver muitas tuplas na relação de montagem com os mesmos valores para os atributos de junção, ou se a função de hash não tiver as propriedades de aleatoriedade e uniformidade. De qualquer forma, algumas das partições terão mais tuplas do que a média, enquanto outras terão menos; o particionamento, então, é considerado **inclinado**.

Podemos lidar com uma pequena quantidade de inclinação aumentando o número de partições, de modo que o tamanho esperado de cada partição (incluindo o índice de hash sobre a partição) seja um pouco menor que o tamanho da memória. O número de partições, portanto, aumenta em um valor pequeno, chamado **fator de fudge**, que normalmente é cerca de 20% do número de partições de hash calculadas, conforme descrevemos na seção "Junção hash".

Mesmo que, usando um fator de fudge, formos conservadores sobre os tamanhos das partições, os estouros ainda podem ocorrer. Os estouros da tabela de hash podem ser tratados por *solução de estouro* ou *impedimento de estouro*. A **solução de estouro** é realizada durante a fase de montagem, se um estouro de índice de hash for detectado. A solução de estouro prossegue desta maneira: se H_{s_i}, para qualquer *i*, for muito grande, ele é particionado ainda mais em partições menores, usando uma função de hash diferente. De modo semelhante, H_{r_i} também é particionado usando a nova função de hash, e somente as tuplas nas partições que combinam precisam ser verificadas.

Ao contrário, o **impedimento de estouro** realiza o particionamento cuidadosamente, de modo que os estouros nunca ocorrem durante a fase de montagem. No impedimento de estouro, a relação de montagem s é inicialmente particionada em muitas partições pequenas, e depois algumas partições são combinadas de modo que cada uma caiba na memória. A relação de sonda r é particionada da mesma maneira que as partições combinadas sobre *s*, mas os tamanhos de H_{r_i} não importam.

Se uma grande quantidade de tuplas em s tiver o mesmo valor para os atributos de junção, as técnicas de solução e impedimento podem falhar em algumas partições. Nesse caso, em vez de criar um índice de hash na memória e usar uma junção de loop aninhado para unir as partições, podemos usar outras técnicas de junção, como a junção de loop aninhado em bloco, sobre essas partições.

Custo da junção hash

Agora, considere o custo de uma junção hash. Nossa análise considera que não existe estouro da tabela de hash. Primeiro, consideramos o caso em que o particionamento recursivo não é exigido.

- O particionamento das duas relações r e s exige uma leitura completa das duas relações, com uma escrita subseqüente. Essa operação exige $2(b_r + b_s)$ transferências de bloco, onde b_r e b_s indicam o número de blocos contendo registros de relações r e s, respectivamente. As fases de montagem e sonda lêem cada uma das partições uma vez, exigindo outras $b_r + b_s$ transferências de bloco. O número de blocos ocupados pelas partições poderia ser ligeiramente maior que $b_r + b_s$, como resultado de blocos parcialmente preenchidos. O acesso de blocos parcialmente preenchidos pode aumentar uma sobrecarga de no máximo $2n_h$ para cada uma das relações, pois cada uma das n_h partições poderia ter um bloco parcialmente preenchido que precisa ser escrito e lido de volta. Assim, estima-se que uma junção hash exige

$$3(b_r + b_s) + 4n_h$$

transferências de bloco. A sobrecarga $4n_h$ normalmente é muito pequena em comparação com $b_r + b_s$, e pode ser ignorada.

- Supondo que b_b blocos são alocados para o buffer de entrada e cada buffer de saída, o particionamento exige um total de $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$ buscas. As fases de montagem e sonda exigem apenas uma busca para cada uma das n_h partições de cada relação, pois cada partição pode ser lida seqüencialmente. A junção hash, portanto, exige $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$ buscas.

Agora, considere o caso em que o particionamento recursivo é exigido. Cada passada reduz o tamanho de cada uma das partições por um fator esperado de $M - 1$; e as passadas são repetidas até que cada partição tenha o tamanho de no máximo M blocos. O número esperado de passadas exigidas para particionar s é, portanto, $\lceil \log_{M-1}(b_s) - 1 \rceil$.

- Como, em cada passada, cada bloco de s é lido e escrito, o total de transferências de bloco para particionamento de s é $2b_s \lceil \log_{M-1}(b_s) - 1 \rceil$. O número de passadas para particionar r é o mesmo número de passadas para particionar s, e por isso estima-se que a junção deverá exigir

$$2(b_r + b_s)\lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s$$

transferências de bloco.

- Novamente supondo que b_b blocos são alocados para a colocação de cada partição em buffer, e ignorando o número relativamente pequeno de buscas durante a fase de montagem e sonda, a junção hash com particionamento recursivo exige

$$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)\lceil \log_{M-1}(b_s) - 1 \rceil$$

buscas de disco.

Considere, por exemplo, a junção *cliente* ⋈ *depositante*. Com um tamanho de memória de 20 blocos, *depositante* pode ser particionado em cinco partições, cada uma com tamanho de 20 blocos, cujo tamanho caberá na memória. Somente uma passada é necessária para o particionamento. A relação *cliente* é semelhantemente particionada em cinco partições, cada uma com tamanho 80. Ignorando o custo de escrever blocos parcialmente preenchidos, o custo é de $3(100 + 400) = 1500$ transferências de bloco. Existe memória suficiente para alocar 3 buffers para a entrada e cada uma das 5 saídas durante o particionamento, levando a $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$ buscas.

A junção hash pode ser melhorada se o tamanho da memória principal for grande. Quando a entrada de montagem inteira pode ser mantida na memória principal, n_h pode ser definido como 0; depois, o algoritmo de junção hash executa rapidamente, sem particionar as relações em arquivos temporários, independente do tamanho da entrada de sonda. A estimativa de custo diminui para $b_r + b_s$ transferências de bloco e duas buscas.

Junção hash híbrido

O algoritmo de **junção hash híbrido** realiza outra otimização; ele é útil quando os tamanhos de memória são relativamente grandes, mas nem toda a relação de montagem cabe na memória. A fase de particionamento do algoritmo de junção hash precisa de um bloco de memória como buffer para cada partição que é criada, e um bloco de memória como buffer de entrada. Logo, é necessário um total de $n_h + 1$ blocos de memória para particionar as duas relações. Se a memória for maior que n_h+1, podemos usar o restante da memória ($M - n_h - 1$ blocos) para colocar em buffer a primeira partição da entrada de montagem (ou seja, H_{s_0}), de modo que não precisará ser escrita e lida novamente. Além do mais, a função de hash é projetada de modo que o índice de hash sobre H_{s_0} caiba em $M - n_h - 1$ blocos, a fim de que, ao final do particionamento de s, H_{s_0} esteja completamente na memória e um índice de hash possa ser montado sobre H_{s_0}.

Quando o sistema particiona r, ele novamente não grava as tuplas de H_{r_0} em disco; em vez disso, enquanto as gera, o sistema as utiliza para sondar o índice de hash residente na

memória sobre H_{s_0} e gerar tuplas de saída da junção. Depois de serem usadas para a sonda, as tuplas podem ser descartadas e, assim, a partição H_{r_0} não ocupa qualquer espaço na memória. Assim, um acesso de escrita e um acesso de leitura foram economizados para cada bloco de H_{r_0} e H_{s_0}. O sistema escreve tuplas nas outras partições normalmente, e as junta depois. As economias da junção hash híbrido podem ser significativas se a entrada de montagem for apenas ligeiramente maior do que a memória.

Se o tamanho da relação de montagem for b_s, n_h será aproximadamente igual a b_s/M. Assim, a junção hash híbrido é mais útil se $M >> b_s/M$, ou $M >> \sqrt{b_s}$, onde a notação >> indica *muito maior que*. Por exemplo, suponha que o tamanho de bloco seja 4 kilobytes, e o tamanho da relação de montagem seja 1 gigabyte. Então, o algoritmo de junção hash híbrido é útil se o tamanho da memória for significativamente maior do que 2 megabytes; tamanhos de memória superiores a 100 megabytes são muito comuns nos computadores de hoje.

Considere a junção *cliente* ⋈ *depositante* novamente. Com um tamanho de memória de 25 blocos, *depositante* pode ser particionado em cinco partições, cada uma com tamanho de 20 blocos, e a primeira das partições da relação de montagem pode ser mantida na memória. Ela ocupa 20 blocos de memória; um bloco é para a entrada e um bloco de cada é para a inclusão em buffer das outras quatro partições. A relação *cliente* pode ser particionada da mesma forma em cinco partições, cada uma com tamanho de 80, sendo que a primeira delas o sistema usa imediatamente para sonda, em vez de escrever e ler de volta. Ignorando o custo da escrita parcial de blocos preenchidos, o custo é 3 (80 + 320) + 20 + 80 = 1300 transferências de bloco, em vez de 1.500 transferências de bloco sem a otimização do hashing híbrido. Porém, nesse caso, o tamanho do buffer para a entrada e para cada partição escrita em disco diminui para 1 bloco, aumentando o número de buscas para perto do número de acessos de bloco, e daí o custo total. Assim, nesse caso, é melhor usar a junção hash comum com um tamanho de buffer b_b maior do que usar a junção hash híbrida. Porém, com tamanhos de memória muito maiores, aumentar b_b além de algum ponto oferece um benefício menor, e a memória restante pode ser usada para implementar a junção hash híbrido.

Junções complexas

Loop aninhado e junção de loop aninhado em blocos podem ser usados independente das condições de junção. As outras técnicas de junção são mais eficientes do que a junção de loop aninhado e suas variantes, mas só podem tratar de condições de junção simples, como junções naturais e equijunções. Podemos implementar junções com condições de junção complexas, como conjunções e disjunções, usando as técnicas de junção eficientes, se aplicarmos as técnicas desenvolvidas na seção "Implementação de seleções complexas" para o tratamento de seleções complexas.

Considere a seguinte junção com uma condição conjuntiva:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s$$

Uma ou mais das técnicas de junção descritas anteriormente podem ser aplicáveis para junções sobre as condições individuais $r \bowtie_{\theta_1} s$; $r \bowtie_{\theta_2} s$, $r \bowtie_{\theta_3} s$, e assim por diante. Podemos calcular a junção geral primeiro calculando o resultado de uma dessas junções mais simples $r \bowtie_{\theta_i} s$; cada par de tuplas no resultado intermediário consiste em uma tupla de r e uma de s. O resultado completo da junção consiste nas tuplas do resultado intermediário que satisfazem as condições restantes

$$\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$

Essas condições podem ser testadas enquanto as tuplas em $r \bowtie_{\theta_i} s$ estão sendo geradas.

Uma junção cuja condição é disjuntiva pode ser calculada desta maneira: Considere

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n} s$$

A junção pode ser calculada como a junção dos registros em junções individuais $r \bowtie_{\theta_i} s$:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \ldots \cup (r \bowtie_{\theta_n} s)$$

A próxima seção descreve algoritmos para calcular a união de relações.

Outras operações

Outras operações relacionais e operações relacionais estendidas – como eliminação de duplicatas, projeção, operações de conjunto, junção externa e agregação – podem ser implementadas conforme esboçado nas seções "Eliminação de duplicatas" a "Agregação".

Eliminação de duplicatas

Podemos implementar a eliminação de duplicatas facilmente pela classificação. Tuplas idênticas aparecerão adjacentes umas às outras durante a classificação, e todas, exceto uma cópia, poderão ser removidas. Com o sort-merge externo, as duplicatas encontradas enquanto uma rodada está sendo criada podem ser removidas antes que a rodada

seja gravada em disco, reduzindo assim o número de transferências de bloco. As duplicatas restantes podem ser eliminadas durante a mesclagem, e a rodada classificada final não possui duplicatas. A estimativa de custo no pior caso para a eliminação de duplicatas é igual à estimativa de custo do pior caso para a classificação da relação.

Também podemos implementar a eliminação de duplicatas pelo hashing, como no algoritmo de junção hash. Primeiro, a relação é particionada com base em uma função de hash sobre a tupla inteira. Depois, cada partição é lida e um índice de hash na memória é construído. Ao construir o índice de hash, uma tupla só é inserida se ainda não estiver presente. Caso contrário, a tupla é descartada. Depois que todas as tuplas na partição tiverem sido processadas, as tuplas no índice de hash são escritas no resultado. A estimativa de custo é igual àquela para o custo de processamento (particionamento e leitura de cada partição) da relação de montagem em uma junção hash.

Devido ao custo relativamente alto da eliminação de duplicatas, a SQL exige uma solicitação explícita pelo usuário para remover duplicatas; caso contrário, as duplicatas são retidas.

Projeção

Podemos implementar a projeção facilmente realizando a projeção sobre cada tupla (o que gera uma relação que poderia ter registros duplicados), e depois removendo os registros duplicados. As duplicatas podem ser eliminadas pelos métodos descritos na seção anterior. Se os atributos na lista de projeção incluírem a chave da relação, não haverá duplicatas; logo, a eliminação de duplicatas não é exigida. A projeção generalizada (que foi discutida na seção "Projeção generalizada" do Capítulo 2) pode ser implementada da mesma maneira que a projeção.

Operações de conjunto

Podemos implementar as operações de *união*, interseção e *diferença-de-conjunto* primeiro classificando as duas relações e depois varrendo uma vez cada uma das relações classificadas para produzir o resultado. Em $r \cup s$, quando uma varredura simultânea das duas relações revelar a mesma tupla nos dois arquivos, somente uma das tuplas será retida. O resultado de $r \cap s$ terá apenas as tuplas que aparecem nas duas relações. Implementamos a *diferença de conjuntos*, $r - s$, de modo semelhante, retendo as tuplas em r apenas se estiverem ausentes em s.

Para todas essas operações, somente uma varredura das duas relações de entrada será necessária, de modo que o custo é $b_r + b_s$ transferências de bloco se as relações forem classificadas na mesma ordem. Assumindo um pior caso de um buffer de bloco para cada relação, um total de $b_r + b_s$ buscas de disco seria necessário além de $b_r + b_s$ transferências de bloco. O número de buscas pode ser reduzido pela alocação de blocos de buffer extras.

Se as relações não forem classificadas inicialmente, o custo da classificação terá de ser incluído. Qualquer ordem de classificação pode ser usada na avaliação das operações de conjunto, desde que as duas entradas tenham essa mesma ordem de classificação.

O hashing oferece outra maneira de implementar essas operações de conjunto. A primeira etapa em cada caso é particionar as duas relações pela mesma função de hash e, com isso, criar as partições $H_{r_0}, H_{r_1}, \ldots, H_{r_{n_h}}$ e $H_{s_0}, H_{s_1}, \ldots, H_{s_{n_h}}$. Dependendo da operação, o sistema, em seguida, realiza estas etapas sobre cada partição $i = 0, 1, \ldots, n_h$:

- $r \cup s$
  1. Monte um índice de hash na memória sobre H_{r_i}.
  2. Acrescente as tuplas em H_{s_i} ao índice de hash, mas somente se ainda não estiverem presentes.
  3. Acrescente as tuplas no índice de hash ao resultado.
- $r \cap s$
  1. Monte um índice de hash na memória sobre H_{r_i}.
  2. Para cada tupla em H_{s_i}, sonde o índice de hash e envie a tupla ao resultado somente se já estiver presente no índice de hash.
- $r - s$
  1. Monte um índice de hash na memória sobre H_{r_i}.
  2. Para cada tupla em H_{s_i}, sonde o índice de hash e, se a tupla estiver presente no índice de hash, exclua-a do índice de hash.
  3. Acrescente as tuplas restantes no índice de hash ao resultado.

Junção externa

Pense nas *operações de junção externa* descritas na seção "Junção externa" do Capítulo 2. Por exemplo, a junção externa natural à esquerda *cliente* ⟕ *depositante* contém a junção de *cliente* e *depositante* e, além disso, para cada tupla *cliente* t que não possua uma tupla correspondente em *depositante* (ou seja, onde *nome_cliente* não está em *depositante*), a seguinte tupla t_1 é acrescentada ao resultado. Para todos os atributos no esquema de *cliente*, a tupla t_1 tem os mesmos valores da tupla t. Os atributos restantes (do esquema de *depositante*) da tupla t_1 contêm o valor nulo.

Podemos implementar as operações de junção externa usando uma dentre duas estratégias:

1. Calcular a junção correspondente e depois acrescentar outras tuplas ao resultado da junção para

chegar ao resultado da junção externa. Considere a operação de junção externa esquerda e duas relações: $r(R)$ e $s(S)$. Para avaliar $r ⟕_\theta s$, primeiro calculamos $r ⋈_\theta s$ e salvamos o resultado como a relação temporária q_1. Em seguida, calculamos $r - \Pi_R(q_1)$ para obter as tuplas em r que não participam da junção θ. Podemos usar qualquer um dos algoritmos a fim de calcular as junções, a projeção e a diferença de conjunto descritas anteriormente para calcular as junções externas. Preenchemos cada uma dessas tuplas com valores nulos para os atributos de s e as acrescentamos a $q_1$para obter o resultado da junção externa.

A operação de junção externa direita $r ⟖_\theta s$ é equivalente a $s ⟕_\theta r$ e, portanto, pode ser implementada de modo simétrico a junção externa esquerda. Podemos implementar a operação junção externa completa $r ⟗_\theta s$ calculando a junção $r ⋈ s$ e depois acrescentando as tuplas extras das operações de junção externa esquerda e direita, como antes.

2. Modificar os algoritmos de junção. É fácil estender os algoritmos de junção de loop aninhado para calcular a junção externa esquerda: as tuplas na relação externa que não combinam com qualquer tupla na relação interna são escritas na saída depois de serem preenchidas com valores nulos. Porém, é difícil estender a junção de loop aninhado para calcular a junção externa completa.

As junções externas naturais e as junções externas com uma condição de equijunção podem ser calculadas por extensões dos algoritmos de junção merge e junção hash. A junção merge pode ser estendida para calcular a junção externa completa da seguinte maneira: quando o merge das duas relações está sendo feito, as tuplas em qualquer relação que não combinam com qualquer tupla na outra relação podem ser preenchidas com nulos e escritas na saída. De modo semelhante, podemos estender a junção merge para calcular as junções externas à esquerda e à direita escrevendo tuplas que não correspondem (preenchidas com nulos) a partir de apenas uma das relações. Como as relações são classificadas, é fácil detectar se uma tupla combina ou não com quaisquer tuplas da outra relação. Por exemplo, quando uma junção merge de *cliente* e *depositante* é feita, as tuplas são lidas em ordem classificada de *nome_cliente*, e é fácil verificar, para cada tupla, se existe uma tupla correspondente na outra.

As estimativas de custo para implementar junções externas usando o algoritmo de junção merge são as mesmas utilizadas para a junção correspondente. A única diferença está no tamanho do resultado e, portanto, nas transferências de bloco para escrita, que não consideramos em nossas estimativas de custo anteriores.

Você realizará a extensão do algoritmo de junção hash para calcular as junções no Exercício 13.13.

Agregação

Pense no operador de agregação $\mathcal{G}$, discutido na seção "Funções agregadas" do Capítulo 2. Por exemplo, a operação

$$^{nome\_agência}\mathcal{G}\mathbf{sum}(saldo)(conta)$$

agrupa tuplas *conta* por agência e calcula o saldo total de todas as contas em cada agência.

A operação de agregação pode ser implementada da mesma maneira que a eliminação duplicada. Usamos a classificação ou o hashing, assim como fizemos para a eliminação de duplicatas, mas com base no agrupamento de atributos (*nome_agência* no exemplo anterior). Porém, em vez de eliminar tuplas com o mesmo valor para o atributo de agrupamento, nós as reunimos em grupos e aplicamos as operações de agregação sobre cada grupo para chegar ao resultado.

A estimativa de custo para implementar a operação de agregação é igual ao custo da eliminação de duplicatas, para funções agregadas como **min**, **max**, **sum**, **count** e **avg**.

Em vez de reunir todas as tuplas em um grupo e depois aplicar as operações de agregação, podemos implementar as operações de agregação **sum**, **min**, **max**, **count** e **avg** diretamente, enquanto os grupos estão sendo construídos. Para o caso de **sum**, **min** e **max**, quando duas tuplas no mesmo grupo são encontradas, o sistema as substitui por uma única tupla contendo a **sum**, **min** ou **max**, respectivamente, das colunas sendo agregadas. Para a operação **count**, ela mantém um contador acumulado para cada grupo para o qual uma tupla foi encontrada. Finalmente, implementamos a operação **avg** calculando a soma e os valores do contador diretamente, e finalmente dividindo a soma pelo contador, para obter a média.

Se todas as tuplas do resultado couberem na memória, as implementações baseadas em classificação e em hash não precisam gravar quaisquer tuplas em disco. Quando as tuplas são lidas, elas podem ser inseridas em uma estrutura de árvore classificada ou em um índice de hash. Quando usamos as técnicas de agregação diretamente, somente uma tupla precisa ser armazenada para cada um dos grupos. Logo, a estrutura de árvore classificada ou o índice de hash cabem na memória, e a agregação pode ser processada com apenas b_r transferências de bloco (e 1 busca), no lugar das 3 b_r transferências (e um pior caso de até 2 b_r buscas) que seriam exigidas em caso contrário.

Avaliação de expressões

Até aqui, estudamos como as operações relacionais individuais são executadas. Agora, consideramos como avaliar uma expressão contendo várias operações. O modo óbvio de avaliar uma expressão é simplesmente avaliar uma operação de cada vez, em uma ordem apropriada. O resultado de cada avaliação é **materializado** em uma relação temporária para uso subseqüente. Uma desvantagem dessa técnica é a necessidade de construir as relações temporárias, que (a menos que sejam pequenas) precisam ser gravadas em disco. Uma técnica alternativa é avaliar várias operações simultaneamente em uma **canalização** (pipeline), com os resultados de uma operação passados para a seguinte, sem a necessidade de armazenar uma relação temporária.

Nas seções "Materialização" e "Canalização", logo a seguir, consideramos as técnicas de *materialização* e de *pipelining*. Veremos que os custos dessas técnicas podem diferir substancialmente, mas também que existem casos em que apenas a técnica de materialização é viável.

Materialização

É mais fácil entender intuitivamente como avaliar uma expressão examinando a representação gráfica da expressão em uma **árvore de operadores**. Considere a expressão

$$\Pi_{nome\_cliente}\,\sigma_{saldo\,<\,2500}\,(conta) \bowtie cliente)$$

na Figura 13.10.

Se aplicarmos a técnica de materialização, começamos pelas operações de nível mais baixo na expressão (na parte de baixo da árvore). Em nosso exemplo, existe apenas uma operação desse tipo: o operador de seleção sobre *conta*. As entradas das operações de nível mais baixo são relações no banco de dados. Executamos essas operações pelos algoritmos que estudamos anteriormente e armazenamos os resultados em relações temporárias. Podemos usar essas relações temporárias para executar as operações no próximo nível superior na árvore, em que as entradas agora são relações temporárias ou relações armazenadas no banco de dados. No nosso exemplo, as entradas da junção são a relação *cliente* e a relação temporária criada pela seleção sobre *conta*. A junção agora pode ser avaliada, criando outra relação temporária.

Repetindo o processo, por fim, avaliaremos a operação na raiz da árvore, dando o resultado final da expressão. Em nosso exemplo, chegamos ao resultado final executando a operação de projeção na raiz da árvore, usando como entrada a relação temporária criada pela junção.

A avaliação conforme descrita é chamada de **avaliação materializada**, pois os resultados de cada operação intermediária são criados (materializados) e depois são usados para avaliação das operações de nível seguinte.

O custo de uma avaliação materializada não é simplesmente a soma dos custos das operações envolvidas. Quando calculamos as estimativas de custo dos algoritmos, ignoramos o custo da gravação do resultado da operação em disco. Para calcular o custo da avaliação de uma expressão conforme fizemos aqui, temos de somar os custos de todas as operações, além do custo de gravar os resultados intermediários em disco. Consideramos que os registros do resultado se acumulam em um buffer e, quando o buffer está cheio, eles são gravados em disco. O número de blocos gravados, b_r, pode ser estimado como n_r / f_r, onde n_r é o número estimado de tuplas na relação de resultados r, e f_r é o *fator de blocagem* da relação de resultado, ou seja, o número de registros de r que caberão em um bloco. Além do tempo de transferência, algumas buscas de disco podem ser exigidas, pois a cabeça de acesso do disco pode ter se movido entre gravações sucessivas. O número de buscas pode ser estimado como $\lceil b_r / b_b \rceil$, onde b_b é o tamanho do buffer de saída (medido em blocos).

O **buffer duplo** (uso de dois buffers, com um continuando a execução do algoritmo enquanto o outro está sendo gravado) permite que o algoritmo execute mais rapidamente, realizando a atividade da CPU em paralelo com a atividade de E/S. O número de buscas pode ser reduzido com a alocação de blocos extras ao buffer de saída, escrevendo-se vários blocos ao mesmo tempo.

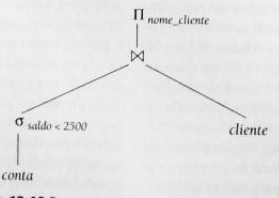


Figura 13.10 Representação gráfica de uma expressão.

Canalização

Podemos melhorar a eficiência da avaliação da consulta reduzindo o número de arquivos temporários que são produzidos. Alcançamos essa redução combinando várias operações relacionais em uma *canalização* de operações, em que os resultados de uma operação são passados adiante para a operação seguinte na canalização. A avaliação conforme descrita é chamada de **avaliação canalizada**. A combinação de operações em uma canalização elimina o custo de ler e escrever relações temporárias.

Por exemplo, considere a expressão $(\Pi_{a1,a2}(r \bowtie s))$. Se a materialização fosse aplicada, a avaliação envolveria a criação de uma relação temporária para manter o resultado da junção, e depois a leitura do resultado para realizar a projeção. Essas operações podem ser combinadas: quando a operação de junção gera uma tupla de seu resultado, ela passa essa tupla imediatamente à operação de projeção, para processamento. Combinando a junção e a projeção, evitamos a criação do resultado intermediário, e, em vez disso, criamos o resultado final diretamente.

Implementação da canalização

Podemos implementar uma canalização construindo uma única operação complexa que combina as operações que constituem a canalização. Embora essa técnica possa ser viável para várias situações que ocorrem com freqüência, em geral é desejável reutilizar o código para operações individuais na construção de uma canalização. Portanto, cada operação na canalização é modelada como um processo ou fluxo de execução (thread) separado dentro do sistema, que apanha um fluxo de tuplas de suas entradas canalizadas e gera um fluxo de tuplas para sua saída. Para cada par de operações adjacentes na canalização, o sistema cria um buffer para manter tuplas sendo passadas de uma operação para a seguinte.

No exemplo da Figura 13.10, todas as três operações podem ser colocadas em uma canalização, que passa os resultados da seleção para a junção à medida que são gerados. Por sua vez, ela passa os resultados da junção para a projeção à medida que são gerados. Os requisitos de memória são baixos, pois os resultados de uma operação não são armazenados por muito tempo. Porém, como um resultado da canalização, as entradas das operações não estão todas disponíveis ao mesmo tempo para processamento.

As canalizações podem ser executadas de duas maneiras:

1. Controladas por demanda
2. Controladas por produtor

Em uma canalização **controlada por demanda**, o sistema faz solicitações repetidas para tuplas da operação no alto da canalização. Toda vez que uma operação recebe uma solicitação para tuplas, ela calcula a próxima tupla (ou tuplas) a ser retornada, e depois retorna essa tupla. Se as entradas da operação não forem canalizadas, as próximas tuplas a serem retornadas podem ser calculadas a partir das relações de entrada, enquanto o sistema registra o que foi retornado até aqui. Se ele tiver algumas entradas canalizadas, a operação também faz solicitações para tuplas de suas entradas canalizadas. Usando as tuplas recebidas de suas entradas canalizadas, a operação calcula tuplas para sua saída, e as passa para o seu pai.

Em uma canalização **controlada por produtor**, as operações não esperam que as solicitações produzam tuplas, mas, em vez disso, geram as tuplas *avidamente*. Cada operação no final de uma canalização gera continuamente as tuplas de saída, e as coloca em um buffer de saída, até que o buffer esteja cheio. Uma operação em qualquer outro nível de uma canalização gera tuplas de saída quando recebe tuplas de entrada de um ponto inferior na canalização, até que seu buffer de saída esteja cheio. Quando a operação usa uma tupla de uma entrada canalizada, ele remove a tupla do seu buffer de entrada. De qualquer forma, quando o buffer de saída está cheio, a operação espera até que sua operação pai remova tuplas do buffer, de modo que o buffer tenha espaço para mais tuplas. Nesse ponto, a operação gera mais tuplas, até que o buffer esteja cheio novamente. A operação repete esse processo até que todas as tuplas de saída tenham sido geradas.

É necessário que o sistema troque entre as operações somente quando um buffer de saída estiver cheio, ou um buffer de entrada estiver vazio e mais tuplas de entrada forem necessárias para gerar mais tuplas de saída. Em um sistema de processamento paralelo, as operações em uma canalização podem ser executadas simultaneamente em processadores distintos (ver Capítulo 21).

O uso da canalização controlada por produtor pode ser considerado como **empurrando** dados de uma árvore de operações de baixo para cima, enquanto o uso da canalização controlada por demanda pode ser considerado como **puxando** os dados de cima em uma árvore de operações. Enquanto as tuplas são geradas *avidamente* na canalização controlada por produtor, elas são geradas **vagarosamente**, por demanda, na canalização controlada por demanda.

Cada operação em uma canalização controlada por demanda pode ser implementada como um **repetidor** que oferece as seguintes funções: *open*( ), *next*( ) e *close*( ). Após uma chamada a *open*( ), cada chamada a *next*( ) retorna a próxima tupla de saída da operação. A implementação da operação, por sua vez, chama *open*( ) e *next*( ) em suas entradas, para apanhar suas tuplas de entrada quando for necessário. A função *close*( ) diz ao repetidor que não são necessárias mais tuplas. O repetidor mantém o es-

tado de sua execução entre as chamadas, de modo que as solicitações *next*( ) sucessivas recebem tuplas de resultado sucessivas.

Por exemplo, para um repetidor implementando a operação select por meio de uma busca linear, a operação *open*( ) inicia uma varredura de arquivo e o estado do repetidor registra o ponto ao qual o arquivo foi varrido. Quando a função *next*( ) é chamada, a varredura de arquivo continua desde o ponto anterior; quando a próxima tupla satisfazendo a seleção é encontrada pela varredura do arquivo, a tupla é retornada após armazenar o ponto em que foi encontrada no estado do repetidor. A operação *open*( ) do repetidor da junção merge abriria suas entradas e, se ainda não estivessem classificadas, ele também as classificaria. Nas chamadas a *next*( ), ele retornaria o próximo par de tuplas correspondentes. A informação de estado chegaria até o ponto em que cada entrada foi varrida.

Os detalhes da implementação dos repetidores ficam para você completar no Exercício prático 13.7. A canalização controlada por demanda normalmente é mais usada do que a canalização controlada por produtor, por ser mais fácil de implementar.

Algoritmos de avaliação para a canalização

Considere uma operação de junção cuja entrada do lado esquerdo está canalizada. Por estar canalizada, a entrada não está disponível ao mesmo tempo para processamento pela operação de junção. Essa indisponibilidade limita a opção do algoritmo de junção a ser usado. A junção merge, por exemplo, não pode ser usada se as entradas não estiverem classificadas, pois não é possível classificar uma relação até que todas as tuplas estejam disponíveis – transformando, assim, a canalização em materialização. Porém, a junção de loop aninhado indexado pode ser utilizada. Quando as tuplas são recebidas para o lado esquerdo da junção, elas podem ser usadas para indexar a relação do lado direito e gerar tuplas no resultado da junção. Esse exemplo ilustra que as escolhas relativas ao algoritmo usado para uma operação e as escolhas referentes à canalização não são independentes.

As restrições sobre os algoritmos de avaliação elegíveis para uso são um fator limitador para a canalização. Como resultado, apesar das aparentes vantagens da canalização, existem casos em que a materialização alcança o menor custo geral. Suponha que a junção de r e s seja exigida, e a entrada r seja canalizada. Se a junção de loop aninhado indexado for usada para dar suporte à canalização, um acesso ao disco pode ser necessário para cada tupla na relação de entrada canalizada. O custo dessa técnica é $n_r * HT_s * (t_S + t_T)$, onde HT_s é a altura do índice sobre s. Com a materialização, o custo da escrita de r seria $b_r * t_T$. Com uma técnica de junção como a junção hash, é possível realizar a junção com um custo de aproximadamente $3(b_r + b_s)$ transferências de bloco mais $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$ buscas (considerando memória suficiente para evitar o particionamento recursivo). Para um n_r grande, a materialização seria menos dispendiosa.

A canalização eficaz requer o uso de algoritmos de avaliação que podem gerar tuplas de saída ao mesmo tempo que as tuplas são recebidas para as entradas da operação. Podemos distinguir entre dois casos:

1. Somente uma das entradas para uma junção é canalizada.
2. As duas entradas para a junção são canalizadas.

Se apenas uma das entradas de uma junção for canalizada, a junção de loop aninhado indexado é uma opção natural. Se as tuplas de entrada canalizadas são classificadas pelos atributos da junção, e a condição de junção for uma equijunção, a junção merge também pode ser usada. A junção hash híbrido também pode ser usada, com a entrada canalizada como relação de sonda. Porém, as tuplas que não estão na primeira partição serão geradas apenas após a relação de entrada canalizada inteira ser recebida. A junção hash híbrido é útil se a entrada não canalizada couber inteiramente na memória, ou se pelo menos a maior parte dessa entrada couber na memória.

Se as duas entradas forem canalizadas, a opção de algoritmos de junção será mais restrita. Se as duas entradas forem armazenadas sobre o atributo junção, e a condição de junção for uma equijunção, a junção merge poderá ser usada. Outra alternativa é a técnica de **junção canalizada**, mostrada na Figura 13.11. O algoritmo considera que as tuplas de entrada para as duas relações de entrada, r e s, estão canalizadas. As tuplas disponíveis para as duas relações são enfileiradas para processamento em uma única fila. Entradas de fila especiais, chamadas End_r e End_s, que servem como marcadores de fim de arquivo, são inseridas na fila após todas as tuplas de r e s (respectivamente) terem sido geradas. Para uma avaliação eficiente, índices apropriados devem ser montados sobre as relações r e s. À medida que as tuplas são acrescentadas a r e s, os índices precisam ser mantidos atualizados.

Resumo

- A primeira ação que o sistema precisa realizar sobre uma consulta é traduzir a consulta para a sua forma interna, que (para sistemas de banco de dados relacionais) normalmente é baseada na álgebra relacional. No processo de geração da forma interna da consulta, o analisador verifica a sintaxe da consulta do usuário, verifica se os

```
done_r := false;
done_s := false;
r := φ;
s := φ;
result := φ;
while not done_r or not done_s do
begin
    if fila vazia, then espera até não estar vazia;
    t := entrada no alto da fila;
    if t = End_r then done_r := true
        else if t = End_s then done_s := true
            else if t é da entrada r
                then
                    begin
                        r := r ∪ {t};
                        result := result ∪ ({t} ⋈ s);
                    end
                else /* t é da entrada s */
                    begin
                        s := s ∪ {t};
                        result := result ∪ (r ⋈ {t});
                    end
end
```

Figura 13.11 Algoritmo de junção canalizada.

nomes de relação que aparecem na consulta são nomes de relações no banco de dados, e assim por diante. Se a consulta foi expressa em termos de uma view, o analisador substitui todas as referências ao nome da view pela expressão da álgebra relacional a fim de calcular a view.

- Dada uma consulta, geralmente existem diversos métodos para calcular a resposta. É responsabilidade do otimizador da consulta transformar a consulta conforme entrada pelo usuário em uma consulta equivalente, que pode ser calculada de forma mais eficiente. O Capítulo 14 aborda a otimização da consulta.
- Podemos processar operações de seleção simples realizando uma varredura linear, fazendo uma busca binária ou utilizando índices. Podemos tratar de seleções complexas calculando uniões e interseções dos resultados de seleções simples.
- Podemos classificar relações maiores do que a memória pelo algoritmo de sort-merge externo.
- As consultas envolvendo uma junção natural podem ser processadas de várias maneiras, dependendo da disponibilidade de índices e da forma de armazenamento físico para as relações.
  - Se o resultado da junção for quase tão grande quanto o produto Cartesiano das duas relações, uma estratégia de *junção de loop aninhado em bloco* pode ser vantajosa.
  - Se os índices estiverem disponíveis, a *junção de loop aninhado indexado* pode ser utilizada.
  - Se as relações estiverem classificadas, uma *junção merge* pode ser desejável. Pode ser vantajoso classificar uma relação antes do cálculo da junção (de modo a permitir o uso da estratégia de junção merge).
  - O algoritmo de *junção hash* divide as relações em várias partes, de modo que cada parte de uma das relações caiba na memória. O particionamento é executado com uma função de hash sobre os atributos da junção, de modo que os pares correspondentes de partições podem ser juntados independentemente.
- As operações de eliminação de duplicatas, projeção e conjunto (união, interseção e diferença) e agregação podem ser feitas por classificação ou por hashing.
- As operações de junção externa podem ser implementadas pelas simples extensões dos algoritmos de junção.
- O hashing e a classificação são duais, no sentido de que qualquer operação – como eliminação de duplicatas, projeção, agregação, junção e junção externa – que possa ser implementada pelo hashing também pode ser implementada pela classificação, e vice-versa; ou seja, qualquer operação que pode ser implementada pela classificação também pode ser implementada pelo hashing.

- Uma expressão pode ser avaliada por meio de materialização, em que o sistema calcula o resultado de cada subexpressão e o armazena no disco, e depois o usa para calcular o resultado da expressão pai.
- A canalização ajuda a evitar a escrita dos resultados de muitas subexpressões em disco, usando os resultados na expressão pai mesmo enquanto estão sendo gerados.

Termos de revisão

- Processamento da consulta
- Primitiva de avaliação
- Plano de execução de consulta
- Plano de avaliação de consulta
- Mecanismo de execução de consulta
- Medidas de custo da consulta
- E/S seqüencial
- E/S aleatória
- Varredura de arquivo
- Busca linear
- Busca binária
- Seleções usando índices
- Caminhos de acesso
- Varreduras de índice
- Seleção conjuntiva
- Seleção disjuntiva
- Índice composto
- Interseção de identificadores
- Classificação externa
- Sort-merge externo
- Rodadas
- Merge de n vias
- Equijunção
- Junção de loop aninhado
- Junção de loop aninhado em bloco
- Junção de loop aninhado indexado
- Junção merge
- Junção sort-merge
- Junção merge híbrida
- Junção hash
  - Montagem
  - Sonda
  - Entrada de montagem
  - Entrada de sonda
  - Particionamento recursivo
  - Estouro da tabela de hash
  - Distorção
  - Fator de fudge
  - Solução de estouro
  - Impedimento de estouro
  - Hash de junção híbrida
- Árvore de operadores
- Avaliação materializada
- Buffer duplo
- Avaliação canalizada
  - Canalização controlada por demanda (vagarosamente, puxando)
  - Canalização controlada por produto (avidamente, empurrando)
  - Repetidor
- Junção canalizada

Exercícios práticos

13.1 Considere a seguinte consulta SQL para o nosso banco de dados bancário:

> **select** *T.nome_agência*
> **from** *agência T*, *agência S*
> **where** *T.ativos* > *S.ativos* **and** *S.cidade_agência* = "Brooklyn"

Escreva uma expressão da álgebra relacional que seja equivalente a essa consulta. Justifique sua escolha.

13.2 Suponha (por simplicidade neste exercício) que somente uma tupla se encaixe em um bloco e a memória mantenha no máximo 3 frames de página. Mostre as rodadas criadas em cada passada do algoritmo de sort-merge, quando aplicadas para classificar as seguintes tuplas sobre o primeiro atributo: (kangaroo, 17), (wallaby, 21), (emu, 1), (wombat, 13), (platypus, 3), (lion, 8), (warthog, 4), (zebra, 11), (meerkat, 6), (hyena, 9), (hornbill, 2), (baboon, 12).

13.3 Considere que as relações r_1 (C,D,E) e r_2 (C,D,E) tenham as seguintes propriedades: r_1 tem 20.000 tuplas, r_2 tem 45.000 tuplas, um bloco pode conter 25 tuplas de r_1 ou 30 tuplas de r_2. Estime o número de transferências de bloco e buscas necessárias, usando cada uma das seguintes estratégias de junção para $r_1 \bowtie r_2$.

a. Junção de loop aninhado
b. Junção de loop aninhado em bloco
c. Junção merge
d. Junção hash

13.4 O algoritmo de junção de loop aninhado indexado descrito na seção "Junção de loop aninhado indexado" pode ser ineficaz se o índice for um índice secundário e se houver várias tuplas com o mesmo valor para os atributos da junção. Por que isso é ineficiente? Descreva uma maneira, usando a classificação, de reduzir o custo da leitura de tuplas da relação interior. Sob que condições esse algoritmo seria mais eficiente do que a junção merge híbrida?

13.5 Considere que r e s sejam relações sem índices, e considere que as relações não estão classificadas. Considerando memória infinita, qual é a maneira com menor custo (em termos de operações de E/S) para calcular $r \bowtie s$? Qual é a quantidade de memória necessária para esse algoritmo?

13.6 Suponha que um índice de árvore B$^+$ sobre *cidade_agência* esteja disponível na relação *agência*, e que nenhum outro índice esteja disponível. Liste as diferentes maneiras de lidar com as seguintes seleções que envolvem negação.

a. $\sigma_{\neg(cidade\_agência < \text{"Brooklyn"})}(agência)$

b. $\sigma_{\neg(cidade\_agência = \text{"Brooklyn"})}(agência)$

c. $\sigma_{\neg(cidade\_agência < \text{"Brooklyn"}) \vee ativos < 5000}(agência)$

13.7 Escreva o pseudocódigo para um repetidor que implementa junção de loop aninhado indexado, em que a relação externa é canalizada. Seu pseudocódigo precisa definir as funções-padrão do repetidor *open*( ), *next*( ) e *close*( ). Mostre que informação de estado o repetidor precisa manter entre as chamadas.

13.8 Crie algoritmos baseados em classificação e baseados em hash para calcular a operação de divisão.

13.9 Qual é o efeito sobre o custo da mesclagem de rodadas se o número de blocos de buffer por rodada for aumentado, enquanto a memória geral disponível permanece fixa para o buffer.

Exercícios

13.10 Por que não é desejável forçar os usuários a fazer uma escolha explícita de uma estratégia de processamento de consulta? Existem casos em que seja desejável para os usuários estarem cientes dos custos das estratégias de processamento de consulta concorrentes? Explique sua resposta.

13.11 Crie uma variante do algoritmo de junção merge híbrida para o caso em que as duas relações não estão fisicamente classificadas, mas ambas têm um índice secundário classificado sobre os atributos da junção.

13.12 Estime o número de transferências de bloco e buscas exigidas pela sua solução do Exercício 13.11 para $r_1 \bowtie r_2$, onde r_1 e r_1 r_2 são definidos no Exercício prático 13.3.

13.13 O algoritmo de junção hash, conforme descrito na seção "Junção hash", calcula a junção natural de duas relações. Descreva como estender o algoritmo de junção hash para calcular a junção externa esquerda natural, a junção externa direita natural e a junção externa completa natural. (Dica: Mantenha informações extras com cada tupla no índice de hash, para detectar se qualquer tupla na relação de sonda combina com a tupla no índice de hash.) Experimente seu algoritmo nas relações *cliente* e *depositante*.

13.14 Escreva o pseudocódigo para um repetidor que implementa uma versão do algoritmo sort-merge em que o resultado do merge final é canalizado para seus consumidores. Seu pseudocódigo precisa definir as funções-padrão do repetidor *open*( ), *next*( ) e *close*( ). Mostre qual informação de estado o repetidor precisa manter entre as chamadas.

13.15 A canalização é usada para evitar a escrita de resultados intermediários em disco. Suponha que você precise classificar a relação r usando sort-merge e realizar a junção merge do resultado com uma relação s já classificada.

a. Descreva como a saída da classificação de r pode ser canalizada para a junção merge sem ser gravada de volta no disco.

b. A mesma idéia se aplica mesmo que as duas entradas da junção merge sejam as saídas das operações sort-merge. Porém, a memória disponível precisa ser compartilhada entre as duas operações merge (o próprio algoritmo de junção merge precisa de muito pouca memória). Qual é o efeito de ter de compartilhar a memória sobre o custo de cada operação sort-merge?

13.16 Suponha que você precise calcular ${}_A\mathcal{G}_{soma(C)}(r)$, além de ${}_{A,B}\mathcal{G}_{soma(C)}(r)$. Descreva como calculá-los juntos usando uma única classificação de r.

Notas bibliográficas

Um processador de consulta precisa analisar instruções na linguagem de consulta e traduzi-las para uma forma interna. A análise das linguagens de consulta difere pouco da análise das linguagens de programação tradicionais. A maioria dos textos sobre compilador – como Aho *et al.* [1986] e Tremblay e Sorenson [1985] – aborda as principais técnicas de análise e apresenta a otimização do ponto de vista de uma linguagem de programação.

Graefe [1993] apresenta um excelente levantamento das técnicas de avaliação de consulta.

Knuth [1973] apresenta uma excelente descrição dos algoritmos de classificação externa, incluindo uma otimização chamada *seleção de substituto*, que pode criar rodadas iniciais que têm (na média) o dobro do tamanho da memória. Estudos mais recentes (Nyberg *et al.* [1995]) mostraram que, devido ao fraco comportamento do cache do processador, a seleção de substituto funciona de maneira pior do que o quicksort na memória para a geração da rodada, invalidando os benefícios da geração de rodadas maiores. Nyberg *et al.* [1995] apresentam um algoritmo

de classificação externo que leva em conta os efeitos do cache do processador. Os algoritmos de avaliação de consulta que levam em conta os efeitos do cache foram bastante estudados; veja, por exemplo, Harizopoulos e Ailamaki [2004].

De acordo com os estudos de desempenho realizados em meados da década de 1970, os sistemas de banco de dados desse período usavam apenas junção de loop aninhado e junção merge. Esses estudos, que foram relacionados ao desenvolvimento do System R, determinaram que ou a junção de loop aninhado ou a junção merge quase sempre forneciam o método de junção ideal (Blasgen e Eswaran 1976]); logo, esses dois foram os únicos algoritmos de junção implementados no System R. Entretanto, o estudo do System R não incluía uma análise dos algoritmos de junção hash. Hoje, as junções hash são consideradas altamente eficientes e bastante utilizadas.

Os algoritmos de junção hash foram desenvolvidos inicialmente para sistemas de banco de dados paralelos. As técnicas de junção hash são descritas em Kitsuregawa *et al.* [1983], e as extensões, incluindo a junção hash, são descritas em Shapiro [1986]. Zeller e Gray [1990] e Davison e Graefe [1994] descrevem as técnicas de junção hash que podem se adaptar à memória disponível, o que é importante em sistemas em que várias consultas podem estar rodando ao mesmo tempo. Graefe *et al.* [1998] descrevem o uso de junções hash e *equipes de hash*, que permitem a canalização de junções hash usando o mesmo particionamento para todas as junções hash em uma seqüência de canalização, no Microsoft SQL Server.

CAPÍTULO 14

Otimização da consulta

Otimização da consulta é o processo de selecionar o plano de avaliação de consulta mais eficiente dentre as muitas estratégias normalmente possíveis para o processamento de determinada consulta, especialmente se esta for complexa. Não esperamos que os usuários escrevam suas consultas de modo que possam ser processadas de forma eficiente. Em vez disso, esperamos que o sistema construa um plano de avaliação de consulta que reduza o custo da avaliação da consulta. É aí que a otimização da consulta entra em ação.

Um aspecto da otimização ocorre no nível da álgebra relacional, em que o sistema tenta encontrar uma expressão que seja equivalente à expressão dada, porém, cuja execução seja mais eficiente. Outro aspecto é a seleção de uma estratégia detalhada para processamento da consulta, como a escolha do algoritmo a ser usado para a execução de uma operação, a escolha dos índices específicos a utilizar, e assim por diante.

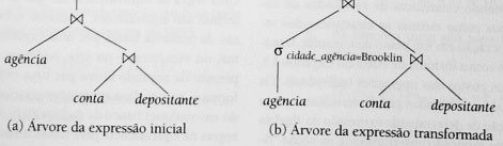
A diferença no custo (em termos do tempo de avaliação) entre uma boa estratégia e uma estratégia ruim normalmente é substancial e pode chegar a várias ordens de grandeza. Logo, vale a pena para o sistema gastar uma quantidade de tempo substancial na seleção de uma boa estratégia para processar uma consulta, mesmo que a consulta seja executada apenas uma vez.

Visão geral

Considere a expressão da álgebra relacional para a consulta "Encontrar os nomes de todos os clientes que tenham uma conta em qualquer agência localizada no Brooklyn",

$$\Pi_{nomecliente}\,(\sigma_{cidadeagência\,=\,\text{"Brooklyn"}}\,(agência \bowtie (conta \bowtie depositante)))$$

Essa expressão constrói uma grande relação intermediária, *agência* ⋈ *conta* ⋈ *depositante*. Porém, estamos interessados em apenas algumas poucas tuplas dessa relação (aque-



(a) Árvore da expressão inicial

(b) Árvore da expressão transformada

Figura 14.1 Expressões equivalentes.

las pertencendo a agências localizadas no Brooklyn), e em apenas um dos seis atributos dessa relação. Como estamos preocupados apenas com as tuplas da relação agência que pertencem a agências localizadas no Brooklyn, não precisamos considerar as tuplas que não têm *cidade_agência* = "Brooklyn". Reduzindo o número de tuplas da relação *agência* que precisamos acessar, reduzimos o tamanho do resultado intermediário. Nossa consulta agora é representada pela expressão da álgebra relacional

$$\Pi_{nomecliente}\left(\left(\sigma_{cidadeagência = \text{"Brooklyn"}}(agência)\right) \bowtie (conta \bowtie depositante)\right)$$

que é equivalente à nossa expressão algébrica original, mas que gera relações intermediárias menores. A Figura 14.1 representa as expressões inicial e transformada.

Dada uma expressão da álgebra relacional, é tarefa do otimizador da consulta gerar um plano de avaliação de consulta que calcule o mesmo resultado que a expressão indicada, com uma maneira menos dispendiosa de gerar o resultado (ou, pelo menos, não muito mais dispendiosa do que a maneira com menor custo).

Para encontrar o plano de avaliação de consulta com menor custo, o otimizador precisa gerar planos alternativos que produzam o mesmo resultado que a expressão dada e escolher o que tiver menor custo. A geração de planos de avaliação de consulta envolve três etapas: (1) gerar expressões que sejam logicamente equivalentes à expressão dada, (2) estimar o custo de cada plano de avaliação e (3) anotar as expressões resultantes de maneiras alternativas para gerar planos de avaliação de consulta alternativos. As etapas (1) e (3) são intercaladas no otimizador de consulta – algumas expressões são geradas e anotadas, depois outras expressões são geradas e anotadas, e assim por diante. A etapa (2) é feita em segundo plano, coletando informações estatísticas sobre as relações, como tamanhos de relação e profundidades de índice, para fazer uma boa estimativa do custo de um plano.

Para implementar a primeira etapa, o otimizador de consulta precisa gerar expressões equivalentes a uma determinada expressão. Ele faz isso por meio de *regras de equivalência* que especificam como transformar uma expressão em outra logicamente equivalente. Descrevemos essas regras na próxima seção.

Na seção "Estimando estatísticas de resultados de expressão", explicamos como estimar as estatísticas dos resultados de cada operação em um plano de consulta. Usando essas estatísticas com a fórmula de custo do Capítulo 13, podemos estimar os custos das operações individuais. Os custos individuais são combinados para determinar o custo estimado da avaliação de determinada expressão da álgebra relacional, conforme esboçado anteriormente na seção "Avaliação de expressões" do Capítulo 13.

Na seção "Escolhas de planos de avaliação", investigamos como escolher um plano de avaliação de consulta. Podemos escolher um com base no custo estimado dos planos. Como o custo é uma estimativa, o plano selecionado não é necessariamente o plano menos dispendioso; porém, desde que as estimativas sejam boas, o plano provavelmente será o menos dispendioso, ou não muito acima dele. Essa otimização, denominada **otimização baseada em custo**, é descrita na seção "Otimização baseada em custo".

Finalmente, as views materializadas ajudam a agilizar o processamento de certas consultas. Na seção "Views materializadas", estudamos como "manter" views materializadas – ou seja, mantê-las atualizadas – e como realizar a otimização da consulta com views materializadas.

Transformação de expressões relacionais

Uma consulta pode ser expressa de várias maneiras diferentes, com diferentes custos de avaliação. Nesta seção, em vez de apanhar a expressão relacional conforme dada, consideramos expressões alternativas, equivalentes.

Duas expressões da álgebra relacional são consideradas **equivalentes** se, em cada instância de banco de dados válida, as duas expressões gerarem o mesmo conjunto de tuplas. (Lembre-se de que uma instância de banco de dados válida é aquela que satisfaz todas as restrições de integridade especificadas no esquema de banco de dados.) Observe que a ordem das tuplas é irrelevante; as duas expressões podem gerar as tuplas em diferentes ordens, mas seriam consideradas equivalentes desde que o conjunto de tuplas fosse o mesmo.

Em SQL, as entradas e saídas são multiconjuntos de tuplas, e uma versão de multiconjunto da álgebra relacional é usada para avaliar consultas SQL. Duas expressões na versão de *multiconjunto* da álgebra relacional são consideradas equivalentes se, em cada banco de dados válido, as duas expressões gerarem o mesmo multiconjunto de tuplas. A discussão neste capítulo é baseada na álgebra relacional. Deixamos as extensões da versão multiconjunto da álgebra relacional para você, como exercícios.

Regras de equivalência

Uma **regra de equivalência** diz que as expressões de duas formas são equivalentes. Podemos substituir uma expressão da primeira forma por uma expressão da segunda forma, ou vice-versa – ou seja, podemos substituir uma expressão da segunda forma por uma expressão da primeira forma –, pois as duas expressões gerariam o mesmo resultado em qualquer banco de dados válido. O otimizador usa as regras de equivalência para transformar expressões em outras expressões logicamente equivalentes.

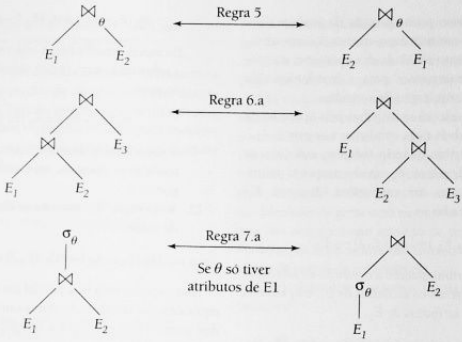


Figura 14.2 Representação gráfica das equivalências.

Agora, listamos diversas regras de equivalência gerais nas expressões da álgebra relacional. Algumas das equivalências listadas aparecem na Figura 14.2. Usamos $\theta, \theta_1, \theta_2$, entre outros, para indicar predicados; L_1, L_2, L_3, entre outros, para indicar listas de atributos e E, E_1, E_2, entre outros, para indicar expressões da álgebra relacional. Um nome de relação r é simplesmente um caso especial de uma expressão da álgebra relacional, e pode ser usado onde quer que E apareça.

1. Operações de seleção conjuntiva podem ser decompostas em uma seqüência de seleções individuais. Esta transformação é considerada como uma cascata de σ.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Operações de seleção são **acumulativas**.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Somente as operações finais em uma seqüência de operações de projeção são necessárias; as outras podem ser omitidas. Essa transformação também pode ser referenciada como uma cascata de Π.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Seleções podem ser combinadas com produtos Cartesianos e junções theta.

   a) $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$

   Essa expressão é apenas a definição da junção theta.

   b) $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Operações de junção theta são acumulativas.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

Na realidade, a ordem dos atributos difere entre o lado da esquerda e o lado da direita, de modo que a equivalência não se mantém se a ordem dos atributos for levada em conta. Uma operação de projeção pode ser acrescentada a um dos lados da equivalência para reordenar atributos corretamente, mas, por simplicidade, omitimos a projeção e ignoramos a ordem do atributo na maior parte dos nossos exemplos.

Lembre-se de que o operador de junção natural é simplesmente um caso especial do operador de junção theta; logo, as junções naturais também são acumulativas.

6. a. As operações de junção natural são **associativas**.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

   b. As junções theta são associativas da seguinte maneira:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

   onde θ_2 envolve atributos de apenas E_2 e E_3. Qualquer uma dessas condições pode ser vazia;

logo, segue-se que a operação de produto Cartesiano (×) também é associativa. A comutatividade e a associatividade das operações de junção são importantes para a reordenação de junção na otimização da consulta.

7. A operação de seleção se distribui pela operação de junção theta sob as duas condições a seguir:
   a. Ela se distribui quando todos os atributos na condição de seleção θ_0 envolvem apenas os atributos de uma das expressões (digamos, E_1) sendo juntadas. $\bowtie$

$$\sigma_{\theta_0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_{\theta} E_2$$

   b. Ela se distribui quando a condição de seleção θ_1 envolve apenas os atributos de E_1, e θ_2 envolve apenas os atributos de E_2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

8. A operação de projeção se distribui pela operação de junção theta sob as seguintes condições:
   a. Considere que L_1 e L_2 sejam atributos de E_1 e E_2, respectivamente. Suponha que a condição de junção θ envolva apenas atributos em $L_1 \cup L_2$. Então,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = (\Pi_{L_1}(E_1)) \bowtie_{\theta} (\Pi_{L_2}(E_2))$$

   b. Considere uma junção $E_1 \bowtie_{\theta} E_2$. Sejam L_1 e L_2 conjuntos de atributos de E_1 e E_2, respectivamente. Seja L_3 atributos de E_1 que estão envolvidos na condição de junção θ, mas não estão em $L_1 \cup L_2$, e seja L_4 atributos de E_2 que estão envolvidos na condição de junção θ, mas não estão em $L_1 \cup L_2$. Então,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_{\theta} (\Pi_{L_2 \cup L_4}(E_2)))$$

9. As operações de conjunto união e interseção são acumulativas.

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

   Diferença de conjunto não é acumulativa.

10. União e interseção de conjunto são associativas.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. A operação de seleção se distribui pelas operações de união, interseção e diferença de conjunto.

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

   De modo semelhante, a equivalência anterior, com – substituído por $\cup$ ou $\cap$, também se mantém. Além do mais,

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

   A equivalência anterior, com – substituído por $\cap$, também se mantém, mas não se – for substituído por $\cup$.

12. A operação de projeção se distribui pela operação de união.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Essa é apenas uma lista parcial das equivalências. Outras equivalências envolvendo operadores relacionais estendidos, como o join externo e a agregação, são discutidas nos exercícios.

Exemplos de transformações

Agora, ilustramos o uso das regras de equivalência. Usamos nosso exemplo de banco com os esquemas de relação:

Esquema_agência = (*nome_agência, cidade_agência, ativos*)
Esquema_conta = (*número_conta, nome_agência, saldo*)
Esquema_depositante = (*nome_cliente, número_conta*)

As relações *agência*, *conta* e *depositante* são instâncias desses esquemas.

Em nosso exemplo, na primeira seção deste capítulo, a expressão

$$\Pi_{nome\_cliente}(\sigma_{cidade\_agência = \text{"Brooklyn"}}(agência \bowtie (conta \bowtie depositante)))$$

foi transformada na seguinte expressão,

$$\Pi_{nome\_cliente}((\sigma_{cidade\_agência = \text{"Brooklyn"}}(agência)) \bowtie (conta \bowtie depositante))$$

que é equivalente à nossa expressão da álgebra original, mas gera relações intermediárias menores. Podemos executar essa transformação usando a regra 7.a. Lembre-se de que a regra simplesmente diz que as duas expressões são equivalentes; ela não diz que uma é melhor do que a outra.

Várias regras de equivalência podem ser usadas, uma após a outra, sobre uma consulta ou sobre partes da consulta. Como uma ilustração, suponha que modifiquemos nossa consulta original para restringir a atenção a clientes que tenham um saldo superior a 1.000. A nova consulta da álgebra relacional é

$$\Pi_{nome\_cliente}\ (\sigma_{cidade\_agência = \text{"Brooklyn"} \wedge saldo > 1.000}\ (agência \bowtie (conta \bowtie depositante)))$$

Não podemos aplicar o predicado de seleção diretamente à relação *agência*, pois o predicado envolve atributos das relações *agência* e *conta*. Porém, podemos primeiro aplicar a regra 6.a (associatividade de junção natural) para transformar a junção $agência \bowtie (conta \bowtie depositante)$ em $(agência \bowtie conta) \bowtie depositante$:

$$\Pi_{nome\_cliente}\ (\sigma_{cidade\_agência = \text{"Brooklyn"} \wedge saldo > 1000}\ ((agência \bowtie conta) \bowtie depositante))$$

Depois, usando a regra 7.a, podemos reescrever nossa consulta como

$$\Pi_{nome\_cliente}\ ((\sigma_{cidade\_agência = \text{"Brooklyn"} \wedge saldo > 1000}\ (agência \bowtie conta)) \bowtie depositante)$$

Vamos examinar a subexpressão de seleção dentro dessa expressão. Usando a regra 1, podemos dividir a seleção em duas seleções, para chegar à seguinte subexpressão:

$$\sigma_{cidade\_agência = \text{"Brooklyn"}}\ (\sigma_{saldo > 1000}\ (agência \bowtie conta))$$

As duas expressões anteriores selecionam tuplas com *cidade_agência* = "Brooklyn" e *saldo* > 1000. Porém, a última forma da expressão oferece uma nova oportunidade para aplicar a regra 7.a. ("realizar seleções mais cedo"), resultando na subexpressão

$$\sigma_{cidade\_agência = \text{"Brooklyn"}}\ (agência) \bowtie \sigma_{saldo > 1000}\ (conta)$$

A Figura 14.3 representa a expressão inicial e a expressão final depois de todas essas transformações. Da mesma forma, também poderíamos ter usado a regra 7.b para chegar diretamente à expressão final, sem usar a regra 1 para dividir a seleção em duas. Na verdade, a própria regra 7.b pode ser derivada a partir das regras 1 e 7.a.

Um conjunto de regras de equivalência é considerado **mínimo** se nenhuma regra puder ser derivada de qualquer combinação das outras. O exemplo anterior ilustra que o conjunto de regras de equivalência na seção anterior não é mínimo. Uma expressão equivalente à expressão original pode ser gerada de diferentes maneiras; o número de maneiras diferentes de gerar uma expressão aumenta quando usamos um conjunto não mínimo de regras de equivalência. Portanto, os otimizadores de consulta usam conjuntos mínimos de regras de equivalência.

Agora, considere a seguinte forma da nossa consulta de exemplo:

$$\Pi_{nome\_cliente}\ ((\sigma_{cidade\_agência = \text{"Brooklyn"}}\ (agência) \bowtie conta) \bowtie depositante)$$

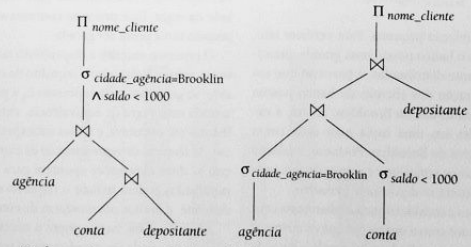
Quando calculamos a subexpressão

$$(\sigma_{cidade\_agência = \text{"Brooklyn"}}\ (agência) \bowtie conta)$$

obtemos uma relação cujo esquema é

(*nome_agência, cidade_agência, ativos, número_conta, saldo*)

Podemos eliminar vários atributos do esquema, aplicando projeções com base nas regras de equivalência 8.a e 8.b. Os únicos atributos que precisamos reter são aqueles que ou aparecem no resultado da consulta ou são necessários para processar operações subsequentes. Eliminando atributos desnecessários, reduzimos a quantidade de colunas do resultado intermediário. Em nosso exemplo, o único atributo



(a) Árvore de expressão inicial (b) Árvore após múltiplas transformações

Figura 14.3 Transformações múltiplas.

que precisamos da junção de *agência* e *conta* é *número_conta*. Portanto, podemos modificar a expressão para

$$\Pi_{nome\_cliente}\,(\Pi_{número\_conta}((\sigma_{cidade\_agência = \text{"Brooklyn"}}\,(agência)) \bowtie conta)) \bowtie depositante)$$

A projeção $\Pi_{número\_conta}$ reduz o tamanho dos resultados intermediários da junção.

Ordenação de junção

Uma boa ordenação das operações de junção é importante para reduzir o tamanho dos resultados temporários; daí a maioria dos otimizadores de consulta prestarem muita atenção à ordem da junção. Como dissemos no Capítulo 2 e na regra de equivalência 6.a, a operação de junção natural é associativa. Assim, para todas as relações r_1, r_2 e r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

Embora essas expressões sejam equivalentes, os custos do seu cálculo podem diferir. Considere novamente a expressão

$$\Pi_{nome\_cliente}\,((\sigma_{cidade\_agência = \text{"Brooklyn"}}\,(agência)) \bowtie conta \bowtie depositante)$$

Poderíamos decidir calcular *conta* ⋈ *depositante* primeiro e depois juntar o resultado com

$$\sigma_{cidade\_agência = \text{"Brooklyn"}}\,(agência)$$

Porém, *conta* ⋈ *depositante* provavelmente será uma relação grande, pois contém uma tupla para cada conta. Ao contrário,

$$\sigma_{cidade\_agência = \text{"Brooklyn"}}\,(agência) \bowtie conta$$

é provavelmente uma relação pequena. Para verificar isso, observamos que, como o banco possui uma grande quantidade de agências bastante distribuídas, é provável que somente uma pequena fração dos clientes do banco possua contas em agências localizadas no Brooklyn. Assim, a expressão anterior resulta em uma tupla para cada conta mantida por um residente do Brooklyn. Portanto, a relação temporária que precisamos armazenar é menor do que teria sido se calculássemos *conta* ⋈ *depositante* primeiro.

Existem outras opções a considerar para avaliar nossa consulta. Não nos importamos com a ordem em que os atributos aparecem em uma junção, pois é fácil mudar a ordem antes de exibir o resultado. Assim, para todas as relações r_1 e r_2,

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

Ou seja, o join natural é acumulativo (regra de equivalência 5).

Usando a associatividade e a comutatividade da junção natural (regras 5 e 6), podemos considerar a reescrita de nossa expressão da álgebra relacional como

$$\Pi_{nome\_cliente}\,(((\sigma_{cidade\_agência = \text{"Brooklyn"}}\,(agência)) \bowtie depositante) \bowtie conta)$$

Ou seja, poderíamos calcular

$$(\sigma_{cidade\_agência = \text{"Brooklyn"}}\,(agência)) \bowtie depositante$$

primeiro e, depois disso, juntar o resultado com *conta*. Porém, observe que não existem atributos em comum entre *Esquema_agência* e *Esquema_depositante*, de modo que a junção é apenas um produto Cartesiano. Se houver b agências no Brooklyn e d tuplas na relação *depositante*, esse produto Cartesiano gera $b * d$ tuplas, uma para cada par possível de tupla depositante e agências (sem considerar se a conta em *depositante* é mantida na agência). Assim, parece que esse produto Cartesiano produzirá uma grande relação temporária. Como resultado, rejeitaríamos essa estratégia. Porém, se o usuário tivesse entrado com a expressão anterior, poderíamos usar a associatividade e a comutatividade da junção natural para transformar essa expressão na expressão mais eficiente usada anteriormente.

Enumeração de expressões equivalentes

Otimizadores de consulta utilizam regras de equivalência para gerar sistematicamente expressões equivalentes para a expressão de consulta dada. Conceitualmente, o processo prossegue da seguinte forma. Dada uma expressão, se qualquer subexpressão combinar com um lado da regra de equivalência, o otimizador gera uma nova expressão em que a subexpressão é transformada para combinar com o outro lado da regra. Esse processo continua até que nenhuma expressão nova possa ser gerada.

O processo anterior é dispendioso tanto em espaço quanto em tempo. Veja como o requisito de espaço pode ser reduzido: se gerarmos uma expressão E_1 a partir da expressão E_2 usando uma regra de equivalência, então E_1 e E_2 são semelhantes em estrutura, e temos subexpressões que são idênticas. As técnicas de representação de expressão que permitem que as duas expressões apontem para subexpressões compartilhadas podem reduzir o requisito de espaço significativamente, e muitos otimizadores de consulta as utilizam.

Além do mais, nem sempre é necessário gerar cada expressão que pode ser gerada com as regras de equivalência. Se um otimizador levar em consideração as estimativas de custo da avaliação, ele pode ser capaz de evitar o exame de

algumas das expressões, como veremos na seção "Escolhas de planos de avaliação". Podemos reduzir o tempo exigido para a otimização usando técnicas como estas.

Estimando estatísticas de resultados de expressão

O custo de uma operação depende do tamanho e de outras estatísticas de suas entradas. Dada uma expressão como $a \bowtie (b \bowtie c)$ para estimar o custo da junção de a com $(b \bowtie c)$, precisamos ter estimativas de estatísticas como o tamanho de $b \bowtie c$.

Nesta seção, primeiro listamos algumas estatísticas sobre relações de banco de dados que são armazenadas nos catálogos do sistema de banco de dados, e depois mostramos como usar as estatísticas para estimar estatísticas sobre os resultados de várias operações relacionais.

Uma coisa que se tornará clara mais adiante nesta seção é que as estimativas não são muito precisas, pois são baseadas em suposições que podem não ser exatas. Portanto, um plano de avaliação de consulta que possui o menor custo de execução estimado pode não ter realmente o menor custo de execução real. Porém, a experiência no mundo real tem mostrado que, mesmo que as estimativas não sejam exatas, os planos com os menores custos estimados normalmente têm custos de execução reais que são os menores ou perto disso.

Informações do catálogo

O catálogo do SGBD armazena a seguinte informação estatística sobre as relações do banco de dados:

- n_r, o número de tuplas na relação r.
- b_r, o número de blocos contendo tuplas da relação r.
- l_r, o tamanho de uma tupla de relação r em bytes.
- f_r, o fator de blocagem da relação r – ou seja, o número de tuplas da relação r que cabem em um bloco.
- $V(A, r)$, o número de valores distintos que aparecem na relação r para o atributo A. Esse valor é igual ao tamanho de $\Pi_A(r)$. Se A é uma chave para a relação r, $V(A,r)$ é n_r.

A última estatística, $V(A,r)$, também pode ser mantida para conjuntos de atributos, se for desejado, em vez de apenas para atributos individuais. Assim, dado um conjunto de atributos, A, $V(A,r)$, A é o tamanho de $\Pi_A(r)$.

Se considerarmos que as tuplas da relação r são armazenadas juntas fisicamente em um arquivo, a seguinte equação se mantém:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

As estatísticas sobre índices, como as alturas dos índices de árvore B^+ e o número de páginas de folha nos índices, também são mantidas no catálogo.

Se quisermos manter estatísticas precisas, então, toda vez que uma relação for modificada, também precisamos atualizar as estatísticas. Essa atualização incorre em uma sobrecarga muito grande. Portanto, a maioria dos sistemas não atualiza as estatísticas sobre cada modificação. Em vez disso, eles atualizam as estatísticas durante os períodos de pouca carga do sistema. Como resultado, as estatísticas utilizadas para escolher uma estratégia de processamento de consulta podem não ser completamente precisas. Porém, se não houver muita atividade de atualização nos intervalos entre as atualizações das estatísticas, as características serão suficientemente precisas para oferecer uma boa estimativa dos custos relativos dos diferentes planos.

A informação estatística mencionada aqui é simplificada. Os otimizadores do mundo real normalmente contêm outras informações estatísticas para melhorar a precisão de suas estimativas de custo dos planos de avaliação. Por exemplo, a maioria dos bancos de dados armazena a distribuição de valores para cada atributo como um **histograma**: nele, os valores para o atributo são divididos em uma série

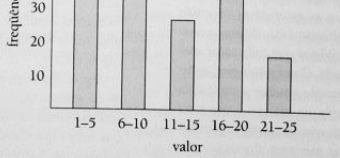


Figura 14.4 Exemplo de histograma.

de intervalos; e com cada intervalo o histograma associa o número de tuplas cujo valor de atributo se encontra nesse intervalo. A Figura 14.4 mostra um exemplo de um histograma para um atributo de valor inteiro que aceita valores no intervalo de 1 a 25.

Os histogramas usados nos sistemas de banco de dados normalmente registram o número de valores distintos em cada intervalo, além do número de tuplas com valores de atributo nesse intervalo.

Como um exemplo de um histograma, o intervalo de valores para um atributo *idade* de uma relação *pessoa* poderia ser dividido em 0–9, 10–19,..., 90–99 (considerando uma idade máxima de 99). Com cada intervalo, armazenamos uma contagem do número de tuplas *pessoa* cujos valores de *idade* se encontram nesse intervalo e o número de valores de idade distintos que se encontram nesse intervalo. Sem tal informação de histograma, um otimizador teria de assumir que a distribuição de valores é uniforme; ou seja, cada intervalo tem a mesma contagem.

Um histograma ocupa um espaço muito pequeno, de modo que histogramas sobre vários atributos diferentes podem ser armazenados no catálogo do sistema. Existem vários tipos de histogramas utilizados nos sistemas de banco de dados. Por exemplo, um **histograma de largura igual** divide o intervalo de valores em intervalos de mesmo tamanho, enquanto um **histograma de profundidade igual** ajusta os limites dos intervalos de modo que cada um tenha o mesmo número de valores.

Estimativa de tamanho da seleção

A estimativa de tamanho do resultado de uma operação de seleção depende do predicado de seleção. Primeiro, consideramos um único predicado de igualdade, depois um único predicado de comparação e, por fim, combinações de predicados.

- $\sigma_{A = a}(r)$: Se consideramos a distribuição uniforme de valores (ou seja, cada valor aparece com mesma probabilidade), o resultado da seleção pode ser estimado como tendo $n_r/V(A, r)$ tuplas, supondo que o valor a apareça no atributo A de algum registro de r. A suposição de que o valor a na seleção aparece em algum registro geralmente é verdadeira, e as estimativas de custo normalmente fazem isso implicitamente. Porém, constantemente não é realista considerar que cada valor aparece com a mesma probabilidade. O atributo *nome_agência* na relação *conta* é um exemplo em que a suposição não é válida. Existe uma tupla na relação *conta* para cada conta. É razoável esperar que as agências grandes tenham mais contas do que as menores. Portanto, certos valores de *nome_agência* aparecem com maior probabilidade do que outros. Apesar do fato de que a suposição de distribuição uniforme normalmente não está correta, essa é uma aproximação razoável da realidade em muitos casos, e nos ajuda a manter nossa apresentação relativamente simples.

  Se um histograma estiver disponível sobre o atributo A, podemos localizar o intervalo que contém o valor a e modificar a estimativa acima mencionada $n_r/V(A, r)$ usando o contador de freqüência para esse intervalo no lugar de n_r, e o número de valores distintos que ocorrem nesse intervalo no lugar de $V(A,r)$.
- $\sigma_{A \leq v}(r)$: Considere uma seleção da forma $\sigma_{A \leq v}(r)$. Se o valor real utilizado na comparação (v) estiver disponível no momento da estimativa de custo, uma estimativa mais precisa poderá ser feita. Os valores mínimo e máximo ($\min(A,r)$ e $\max(A,r)$) para o atributo podem ser armazenados no catálogo. Supondo que os valores são distribuídos uniformemente, podemos estimar o número de registros que satisfarão a condição $A \leq v$ como 0 se $v < \min(A,r)$, como n_r se $v \geq \max(A,r)$, e

$$nr \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(a,r)}$$

  caso contrário.

  Se um histograma estiver disponível sobre o atributo A, podemos chegar a uma estimativa mais precisa; deixamos os detalhes como um exercício para você.

  Em alguns casos, como quando a consulta faz parte de um procedimento armazenado, o valor v pode não estar disponível quando a consulta for otimizada. Nesses casos, iremos supor que aproximadamente metade dos registros satisfarão a condição de comparação. Ou seja, consideramos que o resultado tem $n_r/2$ tuplas; a estimativa pode ser muito imprecisa, mas é o melhor que podemos fazer sem mais informações.
- Seleções complexas:
  - **Conjunção:** Uma *seleção conjuntiva* é uma seleção da forma

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$$

    Podemos estimar o tamanho do resultado de tal seleção: para cada θ_i, estimamos o tamanho da seleção $\sigma_{\theta_i}(r)$, denotada por s_i, conforme descrevemos anteriormente. Assim, a probabilidade de que uma tupla na relação satisfaça a condição de seleção θ_i é s_i / n_r.

    A probabilidade anterior é chamada de **seletividade** da seleção $\sigma_{\theta_i}(r)$. Supondo que as condições sejam *independentes* uma da outra, a probabilidade de que uma tupla satisfaça todas as condições é simplesmente o produto de todas essas probabilidades. Assim,

estimamos o número de tuplas na seleção completa como

$$nr * \frac{s_1 * s_2 * \cdots * s_n}{n_r^n}$$

□ **Disjunção**: Uma *seleção disjuntiva* é uma seleção da forma

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

Uma condição disjuntiva é satisfeita pela união de todos os registros que satisfazem as condições θ_i individuais, simples.

Como antes, considere que s_i / n_r significa a probabilidade de que uma tupla satisfaça a condição θ_i. A probabilidade de que a tupla satisfara a disjunção é, então, 1 menos a probabilidade de que ela *não* satisfará condição alguma:

$$(1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \cdots * (1 - \frac{s_n}{n_r})$$

Multiplicando esse valor por n_r, obtemos o número estimado de tuplas que satisfazem a seleção.

□ **Negação**: Na ausência de nulos, o resultado de uma seleção $\sigma_{\neg\theta}(r)$ é simplesmente as tuplas de r que não estão em $\sigma_\theta(r)$. Já sabemos como estimar o número de tuplas em $\sigma_\theta(r)$. O número de tuplas em $\sigma_{\neg\theta}(r)$ é, portanto, estimado como sendo $n(r)$ menos o número estimado de tuplas em $\sigma_\theta(r)$.

Podemos levar em conta os nulos estimando o número de tuplas para as quais a condição θ seria avaliada como *desconhecido* e subtraindo esse número da estimativa anterior, ignorando nulos. A estimativa desse número exigiria que estatísticas extras fossem mantidas no catálogo.

Estimativa de tamanho de junção

Nesta seção, veremos como estimar o tamanho do resultado de uma junção.

O produto Cartesiano $r \times s$ contém $n_r * n_s$ tuplas. Cada tupla de $r \times s$ ocupa $l_r + l_s$ bytes, dos quais podemos calcular o tamanho do produto Cartesiano.

A estimativa do tamanho de uma junção natural é mais complicada do que a estimativa do tamanho de uma seleção ou de um produto Cartesiano. Suponha que $r(R)$ e $s(S)$ sejam relações.

- Se $R \cap S = \phi$ – ou seja, as relações não possuem atributos em comum –, então $r \bowtie s$ é o mesmo que $r \times s$, e podemos usar nossa técnica de estimativa para produtos Cartesianos.
- Se $R \cap S$ é uma chave para R, então sabemos que uma tupla de s se juntará a no máximo uma tupla de r. Portanto, o número de tuplas em $r \bowtie s$ não é maior do que o número de tuplas em s. O caso em que $R \cap S$ é uma chave para S é simétrico ao caso descrito. Se $R \cap S$ forma uma chave estrangeira de S, referenciando R, o número de tuplas em $r \bowtie s$ é exatamente o mesmo que o número de tuplas em s.
- O caso mais difícil é quando $R \cap S$ não é uma chave nem para R nem para S. Nesse caso, consideramos, como fizemos para as seleções, que cada valor aparece com mesma probabilidade. Considere uma tupla t de r, e considere $R \cap S = \{A\}$. Estimamos que a tupla t produz

$$\frac{n_s}{V(A, s)}$$

tuplas em $r \bowtie s$, pois esse número é o número médio de tuplas em s com determinado valor para os atributos A. Considerando todas as tuplas em r, estimamos que existem

$$\frac{n_r * n_s}{V(A, s)}$$

tuplas em $r \bowtie s$. Observe que, se revertermos os papéis de r e s na estimativa anterior, obtemos uma estimativa de

$$\frac{n_r * n_s}{V(A, r)}$$

tuplas em $r \bowtie s$. Essas duas estimativas diferem se $V(A, r) \neq V(A, s)$. Se essa situação ocorrer, provavelmente haverá tuplas pendentes que não participam da junção. Assim, a menor das duas estimativas provavelmente é a mais exata.

A estimativa anterior do tamanho da junção pode ser muito alta se os valores $V(A, r)$ para A em r tiverem poucos valores em comum com valores $V(A, s)$ para o atributo A em s. Porém, essa situação provavelmente não ocorrerá no mundo real, pois as tuplas pendentes ou não existem ou constituem apenas uma pequena fração das tuplas, na maioria das relações do mundo real.

Mais importante, a estimativa anterior depende da suposição de que cada valor aparece com igual probabilidade. Técnicas mais sofisticadas para estimativa de tamanho precisam ser usadas se essa suposição não for mantida. Por exemplo, se tivéssemos histogramas sobre os atributos de junção das duas relações, e os dois histogramas tivessem os mesmos intervalos, então poderíamos usar a técnica de estimativa anterior dentro de cada intervalo, usando o número de linhas com valores no intervalo no lugar de n_r ou n_s, e o número de valores distintos nesse intervalo, no lugar de $V(A,r)$ ou $V(A,s)$. Depois, somamos as

estimativas de tamanho obtidas para cada intervalo, a fim de chegar à estimativa de tamanho geral. Deixamos como um exercício para o leitor o caso em que as duas relações possuem histogramas sobre o atributo de junção, mas os histogramas possuem intervalos diferentes.

Podemos estimar o tamanho de uma junção theta $r \bowtie_\theta s$ reescrevendo a junção como $\sigma_\theta(r \times s)$ e usando as estimativas de tamanho para produtos Cartesianos junto com as estimativas de tamanho para seleções, que vimos na seção anterior.

Para ilustrar todas essas maneiras de estimar os tamanhos de junção, considere a expressão

$$depositante \bowtie cliente$$

Considere a seguinte informação de catálogo sobre as duas relações:

- $n_{cliente} = 10000$
- $f_{cliente} = 25$, que implica que $b_{cliente} = 10000/25 = 400$.
- $n_{depositante} = 5000$.
- $f_{depositante} = 50$, que implica que $b_{depositante} = 5000/50 = 100$.
- $V(nome\_cliente, depositante) = 2500$, que implica que, na média, cada cliente tem duas contas.

Suponha também que *nome_cliente* em *depositante* seja uma chave estrangeira sobre *cliente*.

No nosso exemplo, de *depositante* $\bowtie$ *cliente*, *nome_cliente* em *depositante* é uma chave estrangeira referenciando *cliente*; logo, o tamanho do resultado é exatamente $n_{depositante}$, que é 5.000.

Agora, vamos calcular as estimativas de tamanho para *depositante* $\bowtie$ *cliente* sem usar informações sobre chaves estrangeiras. Como $V(nome\_cliente, depositante) = 2500$ e $V(nome\_cliente, cliente) = 10000$, nas duas estimativas obtemos 5000 * 10000 / 2500 = 20000 e 5000 * 10000 / 10000 = 5000, e escolhemos a mais baixa. Nesse caso, a mais baixa dessas estimativas é a mesma daquela que calculamos anteriormente pelas informações sobre chaves estrangeiras.

Estimativa de tamanho para outras operações

A seguir, esboçamos como estimar os tamanhos dos resultados de outras operações da álgebra relacional.

- **Projeção:** o tamanho estimado (número de registros ou número de tuplas) de uma projeção na forma $\Pi A(r)$ é V(A,r), pois a projeção elimina duplicatas.
- **Agregação:** o tamanho de ${}_A\mathcal{G}_F(r)$ é simplesmente $V(A,r)$, pois existe uma tupla em ${}_A\mathcal{G}_F(r)$ para cada valor distinto de *A*.
- **Operações de conjunto:** se duas entradas de uma operação de conjunto são seleções sobre a mesma relação, podemos reescrever a operação de conjunto como disjunções, conjunções ou negações. Por exemplo, $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$ pode ser reescrito como $\sigma_{\theta_1 \vee \theta_2}(r)$. De modo semelhante, é possível reescrever interseções como conjuntos e a diferença de conjunto usando negação, contanto que as duas relações participando das operações de conjunto sejam seleções sobre a mesma relação. Podemos, então, usar as estimativas para seleções envolvendo conjunções, disjunções e negação na seção "Estimativa de tamanho da seleção".

  Se as entradas não forem seleções sobre a mesma relação, estimamos os tamanhos desta maneira: o tamanho estimado de $r \cup s$ é a soma dos tamanhos de *r* e *s*. O tamanho estimado de $r \cap s$ é o mínimo dos tamanhos de *r* e *s*. O tamanho estimado de $r - s$ é o mesmo tamanho de *r*. Todas as três estimativas podem ser imprecisas, mas oferecem limites superiores sobre os tamanhos.
- **Junção externa:** o tamanho estimado de $r \mathbin{⟕} s$ é o tamanho de $r \bowtie s$ mais o tamanho de *r*; o tamanho de $r \mathbin{⟖} s$ é simétrico, enquanto o de $r \mathbin{⟗} s$ é o tamanho de $r \bowtie s$ mais os tamanhos de *r* e *s*. Todas as três estimativas podem ser imprecisas, mas oferecem limites superiores sobre os tamanhos.

Estimativa do número de valores distintos

Para seleções, o número de valores distintos de um atributo (ou conjunto de atributos) *A* no resultado de uma seleção, $V(A,\sigma_\theta(r))$, pode ser estimado destas maneiras:

- Se a condição de seleção θ força A para assumir um valor especificado (por exemplo, A=3), $V(A, \sigma_\theta(r)) = 1$.
- Se θ força *A* a assumir um de um conjunto especificado de valores (por exemplo, $(A = 1 \vee A = 3 \vee A = 4)$), depois $V(A, \sigma_\theta(r))$ é definido como o número de valores especificados.
- Se a condição de seleção θ for da forma A op v, onde op é um operador de comparação, $V(A, \sigma_\theta(r))$ é estimado como sendo $V(A,r) * s$, onde s é a seletividade da seleção.
- Em todos os outros casos de seleções, consideramos que a distribuição de valores *A* é independente da distribuição dos valores em que as condições de seleção são especificadas, e usam uma estimativa aproximada de $\min(V(A,r), n_{\sigma_\theta(r)})$. Uma estimativa mais precisa pode ser derivada para esse caso usando teoria de probabilidade, mas essa aproximação funciona muito bem.

Para as junções, o número de valores distintos de um atributo (ou conjunto de atributos) *A* no resultado de uma

junção, $V(A,r \bowtie s)$, pode ser estimado destas maneiras:

- Se todos os atributos em A são de r, $V(A, r \bowtie s)$ é estimado como $\min(V(A,r), n_r \bowtie s)$ e, semelhantemente, se todos os atributos em A são de s, $V(A,r \bowtie s)$ é estimado como sendo $\min(V(A,s), n_r \bowtie s)$.
- Se A contém atributos $A1$ de r e $A2$ de s, então $V(A, r \bowtie s)$ é estimado como

$$\min(V(A1,r) * V(A2\text{-}A1,s), V(A1 - A2,r) * V(A2,s), n_r \bowtie s)$$

Observe que alguns atributos podem estar em $A1$ e também em $A2$, e $A1 - A2$ e $A2 - A1$ denotam, respectivamente, atributos em A que são apenas de r e atributos em A que são apenas de s. Novamente, estimativas mais precisas podem ser derivadas usando a teoria da probabilidade, mas essas aproximações funcionam muito bem.

As estimativas dos valores distintos são simples para projeções: elas são as mesmas em $\Pi A(r)$ e em r. O mesmo se mantém para atributos de agrupamento da agregação. Para os resultados de **sum**, **count** e **average**, podemos considerar, por simplicidade, que todos os valores agregados são distintos. Para $\min(A)$ e $\max(A)$, o número de valores distintos pode ser estimado como $\min(V(A,r), V(G,r))$, onde G denota os atributos de agrupamento. Omitimos os detalhes da estimativa de valores distintos para outras operações.

Escolhas de planos de avaliação

A geração de expressões é apenas parte do processo de otimização, pois cada operação na expressão pode ser implementada com diferentes algoritmos. Um plano de avaliação, portanto, é necessário para definir exatamente que algoritmo deverá ser usado para cada operação e como a execução das operações deve ser coordenada. A Figura 14.5 ilustra um plano de avaliação possível para a expressão da Figura 14.3. Como vimos, vários algoritmos diferentes podem ser usados para cada operação relacional, levando a planos de avaliação alternativos. Além do mais, precisam ser tomadas decisões sobre canalização. Na figura, as bordas das operações de seleção para a operação de junção merge são marcadas como canalizadas; a canalização é viável se as operações de seleção gerarem sua saída classificada sobre os atributos da junção. Elas fariam isso se os índices sobre *agência* e *conta* armazenassem registros com valores iguais para os atributos de índice classificados por *nome_agência*.

Interação de técnicas de avaliação

Uma maneira de escolher um plano de avaliação para uma expressão de consulta é simplesmente escolher para cada operação o algoritmo mais barato para avaliá-la. Podemos escolher qualquer ordem de operações que garanta que as operações mais baixas na árvore sejam executadas antes das mais altas.

No entanto, escolher o algoritmo mais barato para cada operação de maneira independente não é necessariamente uma boa idéia. Embora uma junção merge em determinado nível possa ser mais dispendiosa do que uma junção hash, ela pode oferecer uma saída classificada que torna a avaliação uma operação posterior (como eliminação de duplicatas, interseção ou outra junção merge) menos dispendiosa. De modo semelhante, uma junção de loop aninhado com indexação pode oferecer oportunidades para canalização dos resultados com a próxima operação, e assim pode ser útil mesmo que não seja a forma mais barata de realizar a junção. Para escolher o melhor algoritmo geral, temos de considerar até mesmo algoritmos não ideais para operações individuais.

Assim, além de considerar expressões alternativas para uma consulta, também temos de considerar algoritmos alter-

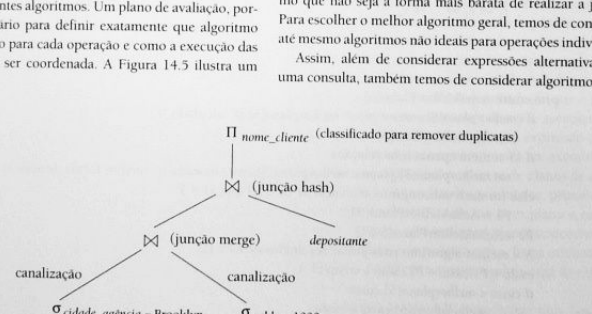


Figura 14.5 Um plano de avaliação.

nativos para cada operação em uma expressão. Podemos usar regras de modo semelhante às regras de equivalência para definir quais algoritmos podem ser usados para cada operação, e se seu resultado pode ser canalizado ou tiver de ser materializado. Podemos usar essas regras para gerar todos os planos de avaliação de consulta para determinada expressão.

Dado um plano de avaliação, podemos estimar seu custo usando estatísticas estimadas pelas técnicas na seção "Estimando estatísticas de resultados de expressão", junto com estimativas de custo para diversos algoritmos e métodos de avaliação descritos no Capítulo 13. Dependendo dos índices disponíveis, certas operações de seleção podem ser avaliadas usando apenas um índice sem acessar a própria relação. Isso ainda deixa o problema de escolher o melhor plano de avaliação para uma consulta. Existem duas técnicas gerais: a primeira pesquisa todos os planos e escolhe o melhor plano em um modelo baseado em custo. A segunda usa heurística para escolher um plano. Discutimos essas técnicas em seguida. Otimizadores de consulta práticos incorporam elementos das duas técnicas.

Otimização baseada em custo

Um **otimizador baseado em busco** gera uma série de planos de avaliação de consulta a partir de determinada consulta, usando as regras de equivalência, e escolhe a única com o menor custo. Para uma consulta complexa, o número de planos de consulta diferentes que são equivalentes a determinado plano pode ser grande. Como ilustração, considere a expressão

$$r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

onde as junções são expressas sem qualquer ordenação. Com $n = 3$, existem 12 ordens de junção diferentes:

$$r_1 \bowtie (r_2 \bowtie r_3) \quad r_1 \bowtie (r_3 \bowtie r_2) \quad (r_2 \bowtie r_3) \bowtie r_1 \quad (r_3 \bowtie r_2) \bowtie r_1$$
$$r_2 \bowtie (r_1 \bowtie r_3) \quad r_2 \bowtie (r_3 \bowtie r_1) \quad (r_1 \bowtie r_3) \bowtie r_2 \quad (r_3 \bowtie r_1) \bowtie r_2$$
$$r_3 \bowtie (r_1 \bowtie r_2) \quad r_3 \bowtie (r_2 \bowtie r_1) \quad (r_1 \bowtie r_2) \bowtie r_3 \quad (r_2 \bowtie r_1) \bowtie r_3$$

Em geral, com n relações, existem $(2(n-1))!/(n-1)!$ ordens de junção diferentes. (Deixamos o cálculo dessa expressão para você realizar no Exercício prático 14.7.) Para junções envolvendo pequenas quantidades de relações, esse número é aceitável; por exemplo, com $n = 5$, o número é 1680. Porém, quando n aumenta, esse número aumenta rapidamente. Com $n = 7$, o número é 665.280; com $n = 10$, o número é maior do que 17,6 bilhões!

Felizmente, não é necessário gerar todas as expressões equivalentes a determinada expressão. Por exemplo, suponha que queiramos encontrar a melhor ordem de junção da forma

$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

que representa todas as ordens de junção onde r_1, r_2 e r_3 são juntadas primeiro (em alguma ordem), e o resultado é juntado (em alguma ordem) com r_4 e r_5. Existem 12 ordens de junção diferentes para calcular $r_1 \bowtie r_2 \bowtie r_3$ e 12 ordens para calcular a junção desse resultado com r_4 e r_5. Assim, parece haver 144 ordens de junção para examinar. Porém, quando tivermos encontrado a melhor ordem de junção para o subconjunto de relações $\{r_1, r_2, r_3\}$, podemos usar essa ordem para outras junções com r_4 e r_5, e podemos ignorar todas as ordens de junção mais dispendiosas de $r_1 \bowtie r_2 \bowtie r_3$. Assim, em vez de 144 escolhas para examinar, precisamos examinar apenas 12 + 12 escolhas.

```
procedure AchaMelhorPlano(S)
    if (melhorplano[S].custo ≠ ∞) /* melhorplano[S] já calculado */
        return melhorplano[S]
    if (S contém apenas uma relação)
        set melhorplano[S].plano e melhorplano[S].custo baseado na melhor forma de acessar S
    else for each subconjunto não vazio S1 de S tal que S1 ≠ S
        P1 = AchaMelhorPlano(S1)
    P2 = AchaMelhorPlano(S-S1)
    A = melhor algoritmo para juntar resultados de P1 e P2
    custo = P1.custo + P2.custo + custo of A
    if custo < melhorplano[S].custo
            melhorplano[S].custo = custo
            melhorplano[S].plano = "executa P1.plan; executa P2.plan;
                                    resultados da junção de P1 e P2 usando A"
    return melhorplano[S]
```

Figura 14.6 O algoritmo de programação dinâmica para otimização por ordem de junção.

Usando essa idéia, é possível desenvolver um algoritmo de *programação dinâmica* para localizar as ordens de junção ideais. Os algoritmos de programação dinâmica armazenam resultados de cálculos e os reutilizam, um procedimento que pode reduzir bastante o tempo de execução. Um procedimento recursivo implementando o algoritmo de programação dinâmica aparece na Figura 14.6.

O procedimento armazena os planos de avaliação que ele calcula em um array associativo *melhorplano*, que é indexado por conjuntos de relações. Cada elemento do array associativo contém dois componentes: o custo do melhor plano de S e o próprio plano. O valor de *melhorplano*[S].*custo* é considerado como sendo inicializado em ∞ se *melhorplano*[S] ainda não foi calculado.

O procedimento primeiro verifica se o melhor plano para calcular a junção de determinado conjunto de relações S já foi calculado (e armazenado no array associativo *melhorplano*); nesse caso, ele retorna o plano já calculado.

Se S contém apenas uma relação, a melhor maneira de acessar S (levando em conta as seleções sobre S, se houver) é registrada em *melhorplano*. Isso pode envolver o uso de um índice para identificar tuplas, e depois apanhar as tuplas (normalmente conhecido como *varredura de índice*), ou varrer a relação inteira (normalmente conhecido como *varredura de relação*).[1]

Caso contrário, o procedimento experimenta cada maneira de dividir S em dois subconjuntos isolados. Para cada divisão, o procedimento encontra recursivamente os melhores planos para cada um dos dois subconjuntos e depois calcula o custo do plano geral usando essa divisão. O procedimento apanha o plano mais barato dentre todas as alternativas para dividir S em dois conjuntos. O plano mais barato e seu custo são armazenados no array *melhorplano* e retornados pelo procedimento. A complexidade de tempo do procedimento pode ser mostrada como $O(3^n)$ (ver Exercício prático 14.8).

Na realidade, a ordem em que as tuplas são geradas pela junção de um conjunto de relações também é importante para encontrar a melhor ordem de junção geral, pois pode afetar o custo de outras junções (por exemplo, se a junção merge for usada). Uma ordem específica de classificação das tuplas é considerada uma **ordem de classificação interessante** se puder ser útil para uma operação posterior. Por exemplo, a geração do resultado de $r_1 \bowtie r_2 \bowtie r_3$ classificado pelos atributos comuns com r_4 ou r_5 pode ser útil, mas a geração classificada pelos atributos comuns apenas a r_1 e r_2 não é. Usar a junção merge para calcular $r_1 \bowtie r_2 \bowtie r_3$ pode ser mais dispendioso do que usar alguma outra técnica de junção, mas pode oferecer uma saída classificada em uma ordem de classificação interessante.

Logo, não é suficiente encontrar a melhor ordem de junção para cada subconjunto do conjunto de n relações dadas. Em vez disso, temos de encontrar a melhor ordem de junção para cada subconjunto, para cada ordem de classificação interessante do resultado da junção para esse subconjunto. O número de subconjuntos de n relações é 2^n. O número de ordens de classificação geralmente não é grande. Assim, cerca de 2^n expressões de junção precisam ser armazenadas. O algoritmo de programação dinâmica para encontrar a melhor ordem de junção pode ser facilmente estendido para lidar com as ordens de classificação. O custo do algoritmo estendido depende do número de ordens interessantes para cada subconjunto das relações; como se descobriu que esse número é pequeno na prática, o custo permanece em $O(3^n)$. Com n=10, esse número é em torno de 59.000, muito melhor do que 17,6 bilhões de ordens de junção diferentes. Mais importante que isso, o armazenamento exigido é muito menor do que antes, pois só precisamos armazenar uma ordem de junção para cada ordem de classificação interessante de cada um dos 1.024 subconjuntos de $r_1, ..., r_{10}$. Embora os dois números ainda aumentem rapidamente com n, as junções que mais ocorrem normalmente possuem menos de 10 relações, e podem ser tratadas com facilidade.

Podemos usar várias técnicas para reduzir ainda mais o custo da pesquisa por uma grande quantidade de planos. Por exemplo, ao examinarmos os planos para uma expressão, podemos terminar depois que examinarmos apenas uma parte da expressão, isso se determinarmos que o plano mais barato para essa parte já é mais dispendioso do que o plano de avaliação mais barato para uma expressão completa examinada anteriormente. De modo semelhante, suponha que determinemos que o modo mais barato de avaliar a subexpressão é mais dispendioso do que o plano de avaliação mais barato para uma expressão completa examinada anteriormente. Então, nenhuma expressão completa envolvendo essa subexpressão precisa ser examinada. Podemos reduzir ainda mais o número de planos de avaliação que precisam ser considerados por inteiro, primeiro fazendo uma escolha heurística de um bom plano e estimando seu custo. Depois, apenas alguns planos concorrentes exigirão uma análise completa do custo. Essas otimizações podem reduzir significativamente a sobrecarga da otimização de consulta.

As complicações da SQL introduzem uma boa dose de complexidade nos otimizadores de consulta. Esboçamos rapidamente como lidar com as subconsultas aninhadas na seção "Otimizando subconsultas aninhadas".

A técnica de otimização que descrevemos se concentra na otimização da ordem de junção. Ao contrário, os otimi-

1. Se um índice contém todos os atributos de uma relação que são usados em uma consulta, é possível realizar uma *varredura apenas de índice*, que recupera do índice os valores de atributo exigidos, sem buscar as tuplas reais.

zadores usados em alguns outros sistemas, como o Microsoft SQL Server, são baseados em regras de equivalência. O benefício do uso de regras de equivalência é que é fácil estender o otimizador com novas regras. Por exemplo, as consultas aninhadas podem ser representadas usando construções estendidas da álgebra relacional, e as transformações das consultas aninhadas podem ser expressas como regras de equivalência. Para que a técnica funcione de modo eficiente, é preciso que haja técnicas eficientes para detectar derivações duplicadas e uma forma de programação dinâmica para evitar a reotimização das mesmas subexpressões. Essa técnica foi pioneira no projeto de pesquisa Volcano. Veja, nas notas bibliográficas, referências contendo mais informações.

Heurística na otimização

Uma desvantagem da otimização baseada em custo é o próprio custo da otimização. Embora o custo da otimização da consulta possa ser reduzido por algoritmos inteligentes, o número de planos de avaliação diferentes para uma consulta pode ser muito grande, e encontrar o plano ideal a partir desse conjunto requer muito esforço computacional. Logo, os otimizadores utilizam a **heurística** para reduzir o custo da otimização.

Um exemplo de uma regra de heurística é a seguinte, para transformar consultas da álgebra relacional:

- Realize operações de seleção o mais cedo possível.

Um otimizador de heurística usaria essa regra sem descobrir se o custo é reduzido por essa transformação. No primeiro exemplo de transformação da seção "Transformação de expressões relacionais", a operação de seleção foi empurrada para uma junção.

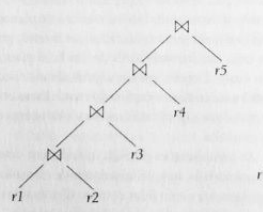
Dizemos que a regra anterior é uma heurística porque ela normalmente, mas nem sempre, ajuda a reduzir o custo. Para ver um exemplo de onde isso pode resultar em um aumento no custo, considere a expressão $\sigma_\theta(r \bowtie s)$, onde a condição θ refere-se apenas a atributos em s. A seleção certamente pode ser realizada antes da junção. Porém, se r for extremamente pequeno em comparação com s, se houver um índice sobre os atributos de junção de s, mas nenhum índice sobre os atributos usados por θ, então provavelmente é uma má idéia realizar a seleção mais cedo, ou seja, diretamente sobre s – porque exigiria uma varredura de todas as tuplas em s. Nesse caso, provavelmente é mais barato calcular a junção usando o índice e depois rejeitar as tuplas que falharem na seleção.

A operação de projeção, como a operação de seleção, reduz o tamanho das relações. Assim, sempre que precisarmos gerar uma relação temporária, será vantajoso aplicar imediatamente quaisquer projeções que são possíveis. Essa vantagem sugere uma companheira para a heurística "realize seleções mais cedo".

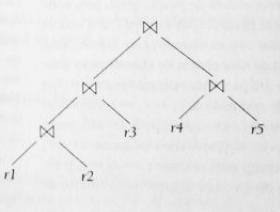
- Realize projeções mais cedo.

Normalmente, é melhor realizar seleções antes das projeções, pois as seleções têm o potencial de reduzir muito os tamanhos das relações, e as seleções permitem o uso de índices para acessar tuplas. Um exemplo semelhante ao que usamos para a heurística de seleção deverá convencê-lo de que essa heurística nem sempre reduz o custo.

Os otimizadores de consulta mais práticos possuem outras heurísticas para reduzir o custo da otimização. Por exemplo, muitos otimizadores de consultas, como o otimizador do System R, não consideram todas as ordens de junção, mas restringem a busca a tipos específicos de ordens de junção. O otimizador do System R considera apenas as ordens de junção em que o operando da direita de cada junção é uma das relações iniciais $r_1, \ldots, r_n$. Essas ordens de junção são consideradas **ordens de junção esquerda profundas**. As ordens de junção esquerda profundas são parti-



(a) Árvore de junção esquerda profunda



(b) Árvore de junção esquerda não profunda

Figura 14.7 Árvores de junção esquerda profundas.

cularmente convenientes para a avaliação canalizada, pois o operando da direita é uma relação armazenada, e somente uma entrada para cada junção é canalizada.

A Figura 14.7 ilustra a diferença entre as árvores de junção esquerda profundas e as árvores de junção esquerda não profunda. O tempo gasto para considerar todas as ordens de junção esquerda profundas é de $O(n!)$, que é muito menos do que o tempo para considerar todas as ordens de junção. Com o uso de otimizações da programação dinâmica, o otimizador do System R pode encontrar a melhor ordem de junção no tempo $O(n2^n)$. Compare esse custo com o tempo $O(3^n)$ necessário para encontrar a melhor ordem de junção geral. O otimizador do System R utiliza a heurística para empurrar as seleções e projeções para baixo na árvore de consulta.

Uma técnica de heurística para reduzir o custo da seleção da ordem de junção, que foi usada originalmente em algumas versões do Oracle, funciona mais ou menos assim: para uma junção de n vias, ela considera n planos de avaliação. Cada plano usa uma ordem de junção esquerda profunda, começando com uma diferente das n relações. A heurística constrói a ordem de junção para cada um dos n planos de avaliação, selecionando repetidamente a "melhor" relação com sua próxima junção, com base em uma classificação dos caminhos de acesso disponíveis. A junção de loop aninhado ou sort-merge é escolhida para cada uma das junções, dependendo dos caminhos de acesso disponíveis. Finalmente, a heurística escolhe um dos n planos de avaliação de uma maneira heurística, com base na redução do número de junções de loop aninhado que não têm um índice disponível na relação interna e no número de junções sort-merge.

As técnicas de otimização de consulta que integram a seleção de heurística e a geração de planos de acesso alternativos foram adotadas em diversos sistemas. A técnica utilizada no System R e em seu sucessor, o projeto Starburst, é um procedimento hierárquico baseado no conceito de bloco aninhado da SQL. As técnicas de otimização baseadas em custo, descritas aqui, são usadas para cada bloco da consulta separadamente. Os otimizadores em diversos produtos de banco de dados, como IBM DB2 e Oracle, são baseados nessa técnica, com extensões para tratar de outras operações, como agregação. Para consultas SQL compostas (usando a operação $\cup$, $\cap$ ou $-$), o otimizador processa cada componente separadamente, e combina os planos de avaliação para formar o plano de avaliação geral.

Muitas aplicações executam a mesma consulta repetidamente, mas com diferentes valores para as constantes. Por exemplo, uma aplicação de banco pode executar uma consulta para encontrar as transações recentes sobre uma conta repetidamente, mas com diferentes valores para o número da conta. Como uma heurística, muitos otimizadores otimizam uma consulta uma vez, com quaisquer valores fornecidos para as constantes quando a consulta foi submetida inicialmente, e mantêm o plano de consulta em cache. Sempre que a consulta é executada novamente, talvez com novos valores para constantes, o plano de consulta em cache é reutilizado (usando novos valores para constantes, é claro). O plano ideal para as novas constantes pode diferir do plano ideal para os valores iniciais, mas, como uma heurística, o plano em cache é reutilizado.

Mesmo com o uso de heurísticas, a otimização de consulta baseada em custo impõe uma sobrecarga substancial ao processamento da consulta. Porém, o custo adicional da otimização de consulta baseada em custo normalmente é mais do que compensada pela economia no tempo de execução da consulta, que é dominada por acessos de disco lentos. A diferença no tempo de execução entre um plano bom e um ruim pode ser muito grande, tornando essencial a otimização da consulta. A economia alcançada é aumentada naquelas aplicações que são executadas regularmente, em que uma consulta pode ser otimizada uma vez, e o plano de consulta selecionado pode ser usado toda vez que a consulta for executada. Portanto, a maioria dos sistemas comerciais inclui otimizadores relativamente sofisticados. As notas bibliográficas dão referências a descrições dos otimizadores de consulta dos sistemas de banco de dados reais.

*Otimizando subconsultas aninhadas***

A SQL conceitualmente trata das subconsultas aninhadas na cláusula where como funções que apanham parâmetros e retornam ou um único valor ou um conjunto de valores (possivelmente um conjunto vazio). Os parâmetros são as variáveis da consulta de nível externo que são usadas na subconsulta aninhada (essas variáveis são chamadas **variáveis de correlação**). Por exemplo, suponha que temos a consulta a seguir.

```
select nome_cliente
from credor
where exists (select *
              from depositante
              where depositante.nome_cliente =
              credor.nome_cliente)
```

Por conceito, a subconsulta pode ser vista como uma função que apanha um parâmetro (aqui, *credor.nome_cliente*) e retorna o conjunto de todos os depositantes com o mesmo nome.

A SQL avalia a consulta geral (conceitualmente) calculando o produto Cartesiano das relações na cláusula from externa e depois testando os predicados na cláusula where para cada tupla no produto. No exemplo anterior, o predicado testa se o resultado da avaliação de subconsulta é vazio.

Essa técnica para avaliar uma consulta com uma subconsulta aninhada é chamada **avaliação correlacionada**. A avaliação correlacionada não é muito eficiente, pois a subconsulta é avaliada separadamente para cada tupla na consulta de nível externo. Isso pode resultar em uma grande quantidade de operações de E/S de disco aleatórias.

Os otimizadores da SQL, portanto, tentam transformar subconsultas aninhadas em junções, onde for possível. Os algoritmos de junção eficientes ajudam a evitar a E/S aleatória e dispendiosa. Onde a transformação não é possível, o otimizador mantém as subconsultas como expressões separadas, otimiza-as separadamene e depois as avalia pela avaliação correlacionada.

Como um exemplo de transformação de uma subconsulta aninhada em uma junção, a consulta do exemplo anterior pode ser reescrita como

```
select nome_cliente
from credor, depositante
where depositante.nome_cliente = credor.nome_cliente
```

(Para refletir corretamente a semântica da SQL, o número de derivações duplicadas não deverá mudar devido à reescrita; a consulta reescrita pode ser modificada para garantir essa propriedade, como veremos em breve.)

No exemplo, a subconsulta aninhada foi muito simples. Em geral, pode não ser possível passar diretamente as relações da subconsulta aninhada para a cláusula **from** da consulta externa. Em vez disso, criamos uma relação temporária que contém os resultados da consulta aninhada *sem* as seleções usando variáveis de correlação da consulta externa, e juntamos a tabela temporária com a consulta de nível externo. Por exemplo, uma consulta no formato

```
select ...
from L1
where P1 and exists (select *
                     from L2
                     where P2)
```

onde P_2 é uma conjunção de predicados simples, pode ser reescrita como

```
create table t1 as
    select distinct V
        from L2
        where P2^1
select ...
from L1, t1
where P1 and P2^2
```

onde P_2^1 contém predicados em P_2 sem as seleções envolvendo variáveis de correlação, e P_2^2 reintroduz as seleções envolvendo variáveis de correlação (com as relações referenciadas no predicado devidamente renomeado). Aqui, V contém todos os atributos que são usados nas seleções com variáveis de correlação na subconsulta aninhada.

Em nosso exemplo, a consulta original teria sido transformada em

```
create table t1 as
    select distinct nome_cliente
    from depositante
select nome_cliente
from credor, t1
where t1.nome_cliente = credor.nome_cliente
```

A consulta que reescrevemos para ilustrar a criação de uma relação temporária pode ser obtida simplificando-se a consulta transformada anterior, considerando que o número de duplicatas de cada tupla não importa.

O processo de substituição de uma consulta aninhada por uma consulta com uma junção (possivelmente com uma relação temporária) é chamado de **descorrelação**.

A descorrelação é mais complicada quando a subconsulta aninhada utiliza agregação, ou quando o resultado da subconsulta aninhada é usado para testar a igualdade, ou quando a condição que liga a subconsulta aninhada à consulta externa é **not exists**, e assim por diante. Não tentamos oferecer algoritmos para o caso geral, e, em vez disso, recomendamos uma consulta aos itens relevantes nas notas bibliográficas.

A otimização de subconsultas aninhadas complexas é uma tarefa difícil, como você pode deduzir pela discussão anterior, e muitos otimizadores realizam apenas uma descorrelação bastante limitada. É melhor evitar o uso de subconsultas aninhadas complexas, onde for possível, pois não podemos ter certeza de que o otimizador de consulta terá sucesso em sua conversão para uma forma que possa ser avaliada de modo eficiente.

Views materializadas**

Quando uma view é definida, normalmente o banco de dados armazena apenas a consulta que a define. Ao contrário, uma **view materializada** é uma view cujo conteúdo é calculado e armazenado. As views materializadas constituem dados redundantes, pois seu conteúdo pode ser deduzido a partir da definição da view e do conteúdo restante do banco de dados. Porém, em muitos casos, é muito menos dispendioso ler o conteúdo de uma view materializada do que calcular o conteúdo da view executando a consulta que define a view.

As views materializadas são importantes para melhorar

o desempenho em algumas aplicações. Considere esta view, que oferece a quantia total de empréstimos em cada agência:

```
create view empréstimo_tot_agência(nome_agência,
empréstimo_total) as
select nome_agência, sum(quantia)
from empréstimo
group by nome_agência
```

Suponha que a quantia total de empréstimos na agência seja solicitada com freqüência (antes de fazer um novo empréstimo, por exemplo). O cálculo da view requer a leitura de cada tupla *empréstimo* pertencente à agência e a soma das quantias de empréstimo, o que pode ser demorado.

Ao contrário, se a definição da view da quantia total de empréstimos fosse materializada, o valor total de empréstimos poderia ser encontrado pesquisando-se uma única tupla na view materializada.

Manutenção de view

Um problema com as views materializadas é que elas precisam ser mantidas atualizadas quando os dados usados na definição da view mudarem. Por exemplo, se o valor de *quantia* de um empréstimo for atualizado, a view materializada se torna incoerente com os dados básicos e deve ser atualizada. A tarefa de manter uma view materializada atualizada com os dados de base é conhecida como **manutenção de view**.

As views podem ser mantidas pelo código escrito manualmente; ou seja, cada parte do código que atualiza o valor de *quantia* de um empréstimo pode ser modificada para também atualizar a quantia total do empréstimo para a agência correspondente.

Outra opção para manter as views materializadas é definir triggers sobre a inserção, exclusão e atualização de cada relação na definição da view. Os triggers precisam modificar o conteúdo da view materializada, para levar em conta a mudança que causou o disparo do trigger. Um modo simplificado de fazer isso é recalcular completamente a view materializada em cada atualização.

Uma opção melhor é modificar apenas as partes afetadas da view materializada, o que é conhecido como **manutenção incremental de view**. Descrevemos como realizar a manutenção incremental de view na próxima seção.

Os sistemas de banco de dados modernos oferecem suporte mais direto para a manutenção incremental de view. Os programadores de sistema de banco de dados não precisam mais definir triggers para a manutenção da view. Em vez disso, quando uma view é declarada como sendo materializada, o sistema de banco de dados calcula o conteúdo da view e o atualiza de forma incremental quando os dados básicos mudam.

A maioria dos sistemas de banco de dados realiza a **manutenção imediata da view**; ou seja, a manutenção incremental da view é realizada assim que ocorre uma atualização, como parte da transação de atualização. Alguns sistemas de banco de dados também admitem a **manutenção de view adiada**, em que a manutenção de view é adiada para uma outra ocasião; por exemplo, as atualizações podem ser coletadas no decorrer de um dia, e as views materializadas podem ser atualizadas à noite. Essa técnica reduz a sobrecarga sobre as transações de atualização. Porém, as views materializadas com manutenção de view adiada podem não ser coerentes com as relações básicas sobre as quais elas são definidas.

Manutenção incremental de view

Para entender como é possível manter as views materializadas de forma incremental, começamos considerando operações individuais, e depois vemos como lidar com uma expressão completa.

As mudanças em uma relação que possam tornar desatualizada uma view materializada são inserções, exclusões e atualizações. Para simplificar nossa descrição, substituímos as atualizações em uma tupla pela exclusão da tupla, seguida da inserção da tupla atualizada. Assim, precisamos considerar apenas inserções e exclusões. As mudanças (inserções e exclusões) feitas a uma relação ou expressão são conhecidas como seu **diferencial**.

Operação de junção

Considere a view materializada $v = r \bowtie s$. Suponha que modifiquemos r inserindo um conjunto de tuplas indicado por i_r. Se o valor antigo de r é indicado por r^{old} e o novo valor de r por r^{new}, $r^{new} = r^{old} \cup i_r$. Agora, o valor antigo da view, v^{old}, é indicado por $r^{old} \bowtie s$, e o novo valor v^{new} é dado por $r^{new} \bowtie s$. Podemos reescrever $r^{new} \bowtie s$ como $(r^{old} \cup i_r) \bowtie s$, que novamente podemos reescrever como $(r^{old} \bowtie s) \cup (i_r \bowtie s)$. Em outras palavras,

$$v^{new} = v^{old} \cup (i_r \bowtie s)$$

Assim, para atualizar a view materializada v, simplesmente precisamos acrescentar as tuplas $i_r \bowtie s$ ao conteúdo antigo da view materializada. Os inserts em s são tratados de uma maneira exatamente simétrica.

Agora, suponha que r seja modificado pela exclusão de um conjunto de tuplas indicado por d_r. Usando o mesmo raciocínio de antes, obtemos

$$v^{new} = v^{old} - (d_r \bowtie s)$$

As exclusões em s são tratadas de modo exatamente simétrico.

Operações de seleção e projeção

Considere uma view $v = \sigma_\theta(r)$. Se modificarmos r inserindo um conjunto de tuplas i_r, o novo valor de v poderá ser calculado como

$$v^{new} = v^{old} \cup \sigma_\theta(i_r)$$

De modo semelhante, se r for modificado pela exclusão de um conjunto de tuplas d_r, o novo valor de v pode ser calculado como

$$v^{new} = v^{old} - \sigma_\theta(d_r)$$

A projeção é uma operação mais difícil de lidar. Considere uma view materializada $v = \Pi_A(r)$. Suponha que a relação r seja sobre o esquema $R = (A, B)$, e r contenha duas tuplas $(a, 2)$ e $(a, 3)$. Então, $\Pi_A(r)$ possui uma única tupla (a). Se excluirmos a tupla $(a,2)$ de r, não poderemos excluir a tupla (a) de $\Pi_A(r)$: se fizéssemos isso, o resultado seria uma relação vazia, enquanto na realidade $\Pi_A(r)$ ainda tem uma única tupla (a). O motivo é que a mesma tupla (a) é derivada de duas maneiras, e a exclusão de uma tupla de r remove apenas uma das maneiras de derivar (a); a outra ainda está presente.

Esse motivo também nos dá a intuição para a solução: para cada tupla em uma projeção como $\Pi_A(r)$, manteremos uma contagem de quantas vezes ela foi derivada.

Quando um conjunto de tuplas d_r for excluído de r, para cada tupla t em d_r, fazemos o seguinte. Considere que $t.A$ indique a projeção de t sobre o atributo A. Encontramos $(t.A)$ na view materializada e diminuímos o contador armazenado com ele por 1. Se o contador se tornar 0, $(t.A)$ será excluída da view materializada.

O tratamento de inserções é relativamente simples. Quando um conjunto de tuplas i_r é inserido em r, para cada tupla t em i_r, fazemos o seguinte. Se $(t.A)$ já estiver presente na view materializada, aumentamos o contador armazenado nela em 1. Se não, acrescentamos $(t.A)$ à view materializada, com o contador definido como 1.

Operações de agregação

As operações de agregação prosseguem de forma semelhante às projeções. As operações agregadas em SQL são count, sum, avg, min e max:

- **count**: considere uma view materializada $v = {}_A\mathcal{G}_{count(B)}(r)$, que calcula o contador do atributo B, depois de agrupar r pelo atributo A.

  Quando um conjunto de tuplas i_r é inserido em r, para cada tupla t em i_r, fazemos o seguinte. Procuramos o grupo $t.A$ na view materializada. Se ele não estiver presente, acrescentamos $(t.A, 1)$ à view materializada. Se o grupo $t.A$ estiver presente, somamos 1 ao contador do grupo.

  Quando um conjunto de tuplas d_r é excluído de r, para cada tupla t em d_r, fazemos o seguinte. Procuramos o grupo $t.A$ na view materializada e subtraímos 1 do contador para o grupo. Se o contador se tornar 0, removemos da view materializada a tupla para o grupo $t.A$.
- **sum**: considere uma view materializada $v = {}_A\mathcal{G}_{sum(B)}(r)$.

  Quando um conjunto de tuplas i_r é inserido em r, para cada tupla t em i_r, fazemos o seguinte. Procuramos o grupo $t.A$ na view materializada. Se ele não estiver presente, acrescentamos $(t.A, t.B)$ à view materializada; além disso, armazenamos um contador de 1 associado a $(t.A, t.B)$, assim como fizemos para a projeção. Se o grupo $t.A$ estiver presente, acrescentamos o valor de $t.B$ ao valor agregado para o grupo e somamos 1 ao contador do grupo.

  Quando um conjunto de tuplas d_r é excluído de r, para cada tupla t em d_r, fazemos o seguinte. Procuramos o grupo $t.A$ na view materializada e subtraímos $t.B$ do valor agregado para o grupo. Também subtraímos 1 do contador para o grupo e, se o contador se tornar 0, excluímos a tupla para o grupo $t.A$ da view materializada.

  Sem manter o valor de contador extra, não poderíamos distinguir um caso em que a soma para um grupo é 0 daquele em que a última tupla no grupo é excluída.
- **avg**: considere uma view materializada $v = {}_A\mathcal{G}_{avg(B)}(r)$.

  A atualização direta da média em uma inserção ou exclusão não é possível, pois depende não apenas da média antiga e da tupla sendo inserida/excluída, mas também do número de tuplas no grupo.

  Em vez disso, para lidar com o caso de **avg**, mantemos os valores agregados de **sum** e **count**, conforme já descrevemos, e calculamos a média como a soma dividida pela contagem.
- **min**, **max**: considere uma view materializada $v = {}_A\mathcal{G}_{min(B)}(r)$. (O caso de **max** é exatamente equivalente.)

  O tratamento de inserções sobre r é simples. A manutenção dos valores agregados **min** e **max** sobre as exclusões pode ser mais dispendiosa. Por exemplo, se a tupla correspondente ao valor mínimo para um grupo for excluída de r, temos de examinar as outras tuplas de r que estão no mesmo grupo para encontrar o novo valor mínimo.

Outras operações

A operação de conjunto *intersecção* é mantida da seguinte maneira. Dada a view materializada $v = r \cap s$, quando uma tupla é inserida em r, verificamos se ela está presente em s e, se estiver, a acrescentamos em v. Se a tupla for excluída de r, nós a excluímos da interseção, se estiver presente. As outras operações de conjunto, *união* e *diferente de conjunto*, são tratadas de modo semelhante; deixamos os detalhes para você.

As junções externas são tratadas mais ou menos da mesma maneira que as junções, mas com algum trabalho extra. No caso da exclusão de r, temos de lidar com tuplas em s que não combinam mais com qualquer tupla em r. No caso da inserção com r, temos de lidar com tuplas em s que não combinam com qualquer tupla em r. Novamente, deixamos os detalhes para você.

Tratamento de expressões

Até aqui, vimos como atualizar de forma incremental o resultado de uma única operação. Para lidar com uma expressão inteira, podemos derivar expressões de modo a calcular a mudança incremental no resultado de cada subexpressão, começando com as menores.

Por exemplo, suponha que queremos atualizar de forma incremental uma view materializada $E_1 \bowtie E_2$ quando um conjunto de tuplas i, for inserido na relação r. Vamos supor que r seja usado apenas em E_1\$. Suponha que o conjunto de tuplas a ser inserido em E_1 seja dado pela expressão D_1. Então, a expressão $D_1 \bowtie E_2$ dá o conjunto de tuplas a ser inserido em $E_1 \bowtie E_2$.

Consulte as notas bibliográficas para ver outros detalhes sobre a manutenção incremental de view com expressões.

Otimização da consulta e views materializadas

A otimização da consulta pode ser realizada tratando-se as views materializadas exatamente como expressões regulares. Porém, as views materializadas oferecem mais oportunidades de otimização:

- Reescrevendo consulta para usar views materializadas:
  Suponha que uma view materializada $v = r \bowtie s$ esteja disponível, e um usuário submeta uma consulta $r \bowtie s \bowtie t$. A reescrita da consulta como $v \bowtie t$ pode oferecer um plano de consulta mais eficiente do que a otimização da consulta conforme submetido. Assim, é tarefa do otimizador de consulta reconhecer quando uma view materializada pode ser usada para agilizar uma consulta.
- Substituindo um uso de uma view materializado pela definição da view:
  Suponha que uma view materializada $v = r \bowtie s$ esteja disponível, mas sem qualquer índice nela, e um usuário submeta uma consulta $\sigma_{A=10}(v)$. Suponha também que s tenha um índice sobre o atributo comum B, e r tenha um índice sobre o atributo A. O melhor plano para essa consulta pode ser substituir v por $r \bowtie s$, que pode levar ao plano de consulta $\sigma_{A=10}(r) \bowtie s$; a seleção e junção podem ser realizadas de forma eficiente pelo uso dos índices sobre $r.A$ e $s.B$, respectivamente. Ao contrário, a avaliação da seleção diretamente sobre v pode exigir uma varredura completa de v, que pode ser mais dispendioso.

As notas bibliográficas indicam trabalhos de pesquisa mostrando como realizar a otimização da consulta de forma eficiente com views materializadas.

Outro problema de otimização relacionado é o da **seleção de view materializada**, ou seja, "qual é o melhor conjunto de views para materializar"? Essa decisão precisa ser feita com base na **sobrecarga** do sistema, que é uma sequência de consultas e atualizações que reflete a carga típica no sistema. Um critério simples seria selecionar um conjunto de views materializadas que reduz o tempo de execução geral da carga de trabalho de consultas e atualizações, incluindo o tempo gasto para manter as views materializadas. Os administradores de banco de dados normalmente modificam esse critério para levar em conta a importância de diferentes consultas e atualizações: a resposta rápida pode ser necessária para algumas consultas e atualizações, mas uma resposta lenta pode ser aceitável para outras.

Os índices são exatamente como views materializadas, no sentido de que também são dados derivados, podem agilizar consultas e podem retardar atualizações. Assim, o problema de **seleção de índice** está bastante relacionado ao da seleção de view materializada, embora mais simples.

Examinamos essas questões com mais detalhes nas seções "Ajuste de índices" e "Usando visões materializadas" do Capítulo 23.

A maioria dos sistemas de banco de dados oferece ferramentas para ajudar o administrador de banco de dados com a seleção de índice e view materializada. Essas ferramentas examinam o histórico das consultas e atualizações, e sugerem índices e views para serem materializados. Microsoft SQL Server Database Tuning Assistant, IBM DB2 Design Advisor e o Oracle SQL Tuning Wizard são exemplos dessas ferramentas.

Resumo

- Dada uma consulta, geralmente existem diversos métodos para calcular a resposta. É responsabilidade do sistema transformar a consulta conforme entrada pelo usuá-

rio em uma consulta equivalente, que pode ser calculada de forma mais eficiente. O processo de localizar uma boa estratégia para processar uma consulta é chamado de *otimização da consulta*.

- A avaliação de consultas complexas envolve muitos acessos ao disco. Como a transferência de dados do disco é lenta em relação à velocidade da memória principal e a CPU do sistema de computador, vale a pena alocar uma quantidade considerável de processamento para escolher um método que reduz os acessos ao disco.
- Existem diversas regras de equivalência que podemos usar para transformar uma expressão em uma equivalente. Usamos essas regras para gerar sistematicamente todas as expressões equivalentes a determinada consulta.
- Cada expressão da álgebra relacional representa uma sequência específica de operações. O primeiro passo na seleção de uma estratégia de processamento de consulta é encontrar uma expressão da álgebra relacional que seja equivalente à expressão indicada e cuja execução deverá custar menos.
- A estratégia que o sistema de banco de dados escolhe para a avaliação de uma operação depende do tamanho de cada relação e da distribuição de valores dentro das colunas. A fim de basear a escolha da estratégia em informações confiáveis, os sistemas de banco de dados podem armazenar estatísticas para cada relação r. Essas estatísticas incluem
  - O número de tuplas na relação r
  - O tamanho de um registro (tupla) da relação r em bytes
  - O número de valores distintos que aparecem na relação r para determinado atributo
- A maioria dos sistemas de banco de dados utiliza histogramas para armazenar o número de valores para um atributo dentro de cada uma das diversas faixas de valores.
- Essas estatísticas nos permitem estimar os tamanhos dos resultados de diversas operações, além do custo de execução dessas operações. A informação estatística sobre as relações é particularmente útil quando vários índices estão disponíveis para ajudar no processamento de uma consulta. A presença dessas estruturas possui uma influência significativa sobre a escolha de uma estratégia de processamento de consulta.
- Planos de avaliação alternativos para cada expressão podem ser gerados por regras de equivalência, e o plano mais barato para todas as expressões pode ser escolhido. Várias técnicas de otimização estão disponíveis para reduzir a quantidade de expressões alternativas e planos que precisam ser gerados.
- Usamos heurísticas para reduzir o número de planos considerados e, assim, reduzir o custo da otimização. As regras da heurística para transformar as consultas da álgebra relacional incluem "Realizar operações de seleção o mais cedo possível", "Realizar projeções mais cedo" e "Evitar produtos Cartesianos".
- As views materializadas podem ser usadas para agilizar o processamento da consulta. A manutenção de view incremental é necessária para atualizar de forma eficiente as views materializadas quando as relações básicas forem modificadas. O diferencial de uma operação pode ser calculado por meio de expressões algébricas envolvendo diferenciais das entradas da operação. Outras questões relacionadas a views materializadas incluem como otimizar consultas usando as views materializadas disponíveis e como selecionar views a serem materializadas.

Termos de revisão

- Otimização da consulta
- Transformação de expressões
- Equivalência de expressões
- Regras de equivalência
  - Comutatividade da junção
  - Associatividade da junção
- Conjunto mínimo de regras de equivalência
- Enumeração de expressões equivalentes
- Estimativa de estatística
- Informações de catálogo
- Estimativa de tamanho
  - Seleção
  - Seletividade
  - Junção
- Histogramas
- Estimativa de valor distinto
- Escolha de planos de avaliação
- Interação de técnicas de avaliação
- Otimização baseada em custo
- Otimização da ordem de junção
  - Algoritmo de programação dinâmica
  - Ordem de junção esquerda profunda
- Otimização de heurística
- Seleção do plano de acesso
- Avaliação correlacionada
- Descorrelação
- Views materializadas
- Manutenção de view materializada
  - Recálculo
  - Manutenção incremental
  - Inserção
  - Exclusão
  - Atualizações
- Otimização da consulta com views materializadas
- Seleção de índice
- Seleção de view materializada

Exercícios práticos

14.1 Mostre que as seguintes equivalências se mantêm. Explique como você pode aplicá-las para melhorar a eficiência de certas consultas:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.

b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, onde θ usa apenas atributos de A.

c. $\sigma_\theta(E_1 \rtimes E_2) = \sigma_\theta(E_1) \rtimes E_2$, onde θ usa apenas atributos de E_1.

14.2 Para cada um dos seguintes pares de expressões, indique exemplos de relações que mostram que as expressões não são equivalentes.

a. $\Pi_A(R - S)$ e $\Pi_A(R) - \Pi_A(S)$.

b. $\sigma_B < 4({}_A\mathcal{G}_{max(B)}(R))$ e ${}_A\mathcal{G}_{max(B)}(\sigma_{B<4}(R))$.

c. Nas expressões anteriores, se as duas ocorrências de max foram substituídas por min, as expressões seriam equivalentes?

d. $(R \rtimes S) \rtimes T$ e $R \rtimes (S \rtimes T)$

Em outras palavras, a junção externa natural não é associativa. (Dica: suponha que os esquemas das três relações sejam $R(a, b1)$, $S(a, b2)$ e $T(a, b3)$, respectivamente.)

e. $\sigma_\theta(E_1 \rtimes E_2)$ e $E_1 \rtimes \sigma_\theta(E_2)$, onde θ usa apenas atributos de E_2.

14.3 A SQL permite relações com duplicatas (Capítulo 3).

a. Defina versões das operações da álgebra relacional básica σ, Π, $\times$, $\bowtie$, $-$, $\cup$ e $\cap$ que atuam sobre relações com duplicatas, em um modo coerente com a SQL.

b. Verifique quais das regras de equivalência de 1 a 7.b se mantêm para a versão multiconjunto da álgebra relacional definida na parte a.

14.4 Considere as relações r_1 (*A,B,C*), r_2 (*C,D,E*) e r_3 (*E,F*), com chaves primárias *A*, *C* e *E*, respectivamente. Suponha que r_1 tenha 1.000 tuplas, r_2 tenha 1.500 tuplas e r_3 tenha 750 tuplas. Estime o tamanho de $r_1 \bowtie r_2 \bowtie r_3$ e dê uma estratégia eficiente para calcular a junção.

14.5 Considere as relações r_1 (*A,B,C*), r_2 (*C,D,E*) e r_3 (*E,F*) do Exercício prático 14.4. Suponha que não existam chaves primárias, exceto o esquema inteiro. Considere que $V(C,r_1)$ seja 900, $V(C,r_2)$ seja 1.000, $V(E,r_2)$ seja 50 e $V(E,r_3)$ seja 100. Suponha que r_1 tenha 1.000 tuplas, r_2 tenha 1.500 tuplas e r_3 tenha 750 tuplas. Estime o tamanho de $r_1 \bowtie r_2 \bowtie r_3$ e dê uma estratégia eficiente para calcular a junção.

14.6 Suponha que um índice de árvore B⁺ sobre *cidadeagência* esteja disponível sobre a relação *agência*, e que nenhum outro índice esteja disponível. Qual seria a melhor maneira de lidar com as seguintes seleções que envolvem negação?

a. $\sigma_{\neg(cidadeagência < \text{"Brooklyn"})}(agência)$

b. $\sigma_{\neg(cidadeagência = \text{"Brooklyn"})}(agência)$

c. $\sigma_{\neg(cidadeagência < \text{"Brooklyn"} \vee ativos < 5000)}(agência)$

14.7 Mostre que, com n relações, existem $(2(n-1))!/(n-1)!$ diferentes ordens de junção.
Dica: uma **árvore binária completa** é aquela em que cada nó interno tem exatamente dois filhos. Use o fato de que o número de árvores binárias completas diferentes com n nós de folha é

$$\frac{1}{n}\binom{2(n-1)}{(n-1)}$$

Se quiser, você pode derivar a fórmula para o número de árvores binárias completas com n nós a partir da fórmula para o número de árvores binárias com n nós. O número de árvores binárias com n nós é

$$\frac{1}{n+1}\binom{2n}{n}$$

Esse número é conhecido como **número de Catalan**, e sua derivação pode ser encontrada em qualquer livro-texto padrão sobre estruturas de dados ou algoritmos.

14.8 Mostre que a ordem de junção de menor custo pode ser calculada no tempo $O(3^n)$. Suponha que você possa armazenar e pesquisar informações sobre um conjunto de relações (como a ordem de junção ideal para o conjunto, e o custo dessa ordem de junção) em tempo constante. (Se você achar este exercício difícil, pelo menos mostre o limite de tempo mais folgado de $O(2^{2n})$.)

14.9 Mostre que, se somente árvores de junção esquerda profunda forem consideradas, como no otimizador do System R, o tempo gasto para encontrar a ordem de junção mais eficiente é em torno de $n2^n$. Suponha que existe apenas uma ordem de classificação interessante.

14.10 Descorrelação:

a. Escreva uma consulta aninhada sobre a relação *conta* para encontrar, para cada agência com nome começando com B, todas as contas com o saldo máximo na agência.

b. Reescreva a consulta anterior, sem usar uma subconsulta aninhada; em outras palavras, descorrelacione a consulta.

c. Dê um procedimento (semelhante ao descrito na seção "Otimizando subconsultas aninhadas") para descorrelacionar essas consultas.

Exercícios

14.11 Suponha que um índice de árvore B⁺ sobre

(*nome_agência, cidadeagência*) esteja disponível sobre a relação *agência*. Qual seria a melhor maneira de lidar com a seguinte seleção?

$$\sigma_{(cidadeagência < \text{"Brooklyn"}) \wedge (ativos < 5000) \wedge (nome\_agência = \text{"Downtown"})} (agência)$$

14.12 Mostre como derivar as seguintes equivalências por uma seqüência de transformações usando as regras de equivalência na seção "Regras de equivalência".

a. $\sigma_{\theta_1 \wedge \theta_2} (E) = \sigma_{\theta_1} (\sigma_{\theta_2} (E))$

b. $\sigma_{\theta_1 \wedge \theta_2} (E_1 \bowtie_{\theta_2} E_2) = \sigma_{\theta_1} (E_1 \bowtie_{\theta_2} (\sigma_{\theta_2} (E_2)))$, onde θ_2 envolve apenas atributos de E_2

14.13 Um conjunto de regras de equivalência é considerado *completo* se, sempre que duas expressões forem equivalentes, uma pode ser derivada da outra por uma seqüência de usos das regras de equivalência. O conjunto de regras de equivalência que consideramos na seção "Regras de equivalência" está completo? Dica: considere a equivalência $\sigma_{3=5}(r) = \{\}$.

14.14 Explique como usar um histograma para estimar o tamanho de uma seleção da forma $\sigma_{A \leq v}(r)$.

14.15 Suponha que duas relações r e s tenham histogramas sobre atributos $r.A$ e $s.A$, respectivamente, mas com diferentes intervalos. Sugira como usar os histogramas para estimar o tamanho de $r \bowtie s$. Dica: divida ainda mais os intervalos de cada histograma.

14.16 Descreva como manter de forma incremental os resultados das seguintes operações, sobre inserções e exclusões.

a. União e diferença de conjunto

b. Junção externa esquerda

14.17 Dê um exemplo de uma expressão definindo uma view materializada e duas situações (conjuntos de estatísticas para as relações de entrada e os diferenciais) de modo que a manutenção de view incremental seja melhor do que o recálculo em uma situação, e o recálculo seja melhor na outra situação.

14.18 Suponha que você obtenha respostas para $r \bowtie s$ classificas sobre um atributo de r e queira somente as K primeiras respostas para algum K relativamente pequeno. Dê uma boa maneira de avaliar a consulta.

a. Quando a junção for sobre uma chave estrangeira de r referenciando s.

b. Quando a junção não é sobre uma chave estrangeira.

14.19 Considere uma relação $r(A,B,C)$, com um índice sobre o atributo A. Dê um exemplo de uma consulta que pode ser respondida com o uso apenas do índice, sem examinar as tuplas na relação. (Planos de consulta que usam apenas o índice, sem acessar a relação real, são chamados planos *somente de índice*.)

Notas bibliográficas

O trabalho pioneiro de Selinger *et al.* [1979] descreve a seleção da via de acesso no otimizador do System R, que foi um dos primeiros otimizadores de consulta relacional. O processamento da consulta do Starburst, descrito em Haas *et al.* [1989], forma a base para a otimização da consulta no IBM DB2.

Graefe e McKenna [1993] descrevem o Volcano, um otimizador de consulta baseado em regra de equivalência, que junto com seu sucessor Cascades Graefe [1995], forma a base da otimização da consulta no Microsoft SQL Server.

Veja, nos Capítulos 27, 28 e 29, mais informações sobre processamento e otimização de consulta no Oracle, IBM DB2 e Microsoft SQL Server, respectivamente.

Estimativa de estatísticas e custo: A estimativa de estatísticas dos resultados da consulta, como tamanho do resultado, é tratada por Ioannidis e Poosala [1995], Poosala e outros [1996] e Ganguly *et al.* [1996], entre outros. Distribuições não uniformes de valores causam problemas para a estimativa do tamanho e custo da consulta. As técnicas de estimativa de custo que utilizam histogramas de distribuições de valor foram propostas para enfrentar o problema. Ioannidis e Christodoulakis [1993], Ioannidis e Poosala [1995] e Poosala *et al.* [1996] apresentam resultados nessa área.

Todos os principais sistemas comerciais de banco de dados utilizam bastante os histogramas para a estimativa de custo. Eles também admitem a amostragem para calcular os histogramas de modo eficiente, sem examinar a relação inteira: um histograma baseado em um subconjunto da relação selecionado aleatoriamente provavelmente será muito parecido com o histograma sobre a relação inteira, se o subconjunto amostrado for grande o suficiente.

Heurística: a pesquisa completa de todos os planos de consulta não é prática para otimizações de junções envolvendo muitas relações. A maioria dos otimizadores utiliza heurísticas para encontrar alguns planos bons, além de usar um algoritmo completo para gerar ordens de junção alternativas. Os algoritmos completos são terminados quando um orçamento de custo é excedido, por exemplo, se o custo de otimização tiver alcançado uma fração significativa do custo de execução estimado.

Otimização de consulta paramétrica: algumas técnicas foram propostas por Ioannidis *et al.* [1992], Ganguly [1998] e Hulgeri e Sudarshan [2003], para otimizar consultas quando a seletividade dos parâmetros de consulta não for conhecida no momento da otimização. Um conjunto de planos – um para cada um de várias seletividades de consulta diferentes – é calculado e armazenado pelo otimiza-

dor, no momento da compilação. Um desses planos é escolhido no momento da execução, com base nas seletividades reais, evitando o custo da otimização completa no momento da execução.

Otimização agregada: Klug [1982] foi um trabalho inicial sobre otimização de expressões da álgebra relacional com funções agregadas. A otimização de consultas com agregação é tratada por Yan e Larson [1995] e Chaudhuri e Shim [1994]. A otimização de consultas contendo junções externas é descrita em Rosenthal e Reiner [1984], Galindo-Legaria e Rosenthal [1992] e Galindo-Legaria [1994].

Otimização dos primeiros K: muitas consultas apanham resultados classificados por alguns atributos e exigem apenas os primeiros *K* resultados para algum *K*. Quando *K* é pequeno, um plano de otimização da consulta que gere o conjunto inteiro de resultados e depois classifique e gere os primeiros *K* será muito ineficiente, pois descartará a maioria dos resultados intermediários que ele calcula. Várias técnicas foram propostas para otimizar as *primeiras K consultas*. Uma técnica é usar planos canalizados que possam gerar os resultados na ordem classificada. Outra técnica é estimar o maior valor sobre os atributos classificados que aparecerá na saída dos primeiros *K*, e introduzir predicados de seleção que eliminam valores maiores. Se outras tuplas além das *K* primeiras forem geradas, elas serão descartadas, e se muito poucas tuplas forem geradas, então a condição de seleção será alterada e a consulta será reexecutada. A otimização das primeiras *K* consultas é tratada em Carey e Kossmann [1998] e Bruno *et al.* [2002].

A linguagem SQL impõe vários desafios para a otimização da consulta, incluindo a presença de duplicatas e nulos e a semântica das subconsultas aninhadas. A extensão da álgebra relacional para duplicatas é descrita em Dayal *et al.* [1982]. A otimização de subconsultas aninhadas é discutida em Kim [1982], Ganski e Wong [1987], Dayal [1987], Seshadri *et al.* [1996] e, mais recentemente, Galindo-Legaria e Joshi [2001].

Minimização de junção: quando as consultas são geradas por meio de visões, mais relações normalmente são juntadas do que é necessário para o cálculo da consulta. Uma coleção de técnicas para minimização de junção foi agrupada sob o nome *otimização de tableau*. A noção de uma tableau foi introduzida por Aho *et al.* [1979b] e Aho et al. [1979a] e foi estendida ainda mais por Sagiv e Yannakakis [1981]. Ullman [1988] e Maier [1983] oferecem uma abordagem de livro-texto sobre tableaux.

Otimização de múltiplas consultas: Sellis [1988] e Roy *et al.* [2000] descrevem a otimização de múltiplas consultas, que é o problema de otimizar a execução de várias consultas como um grupo. Se um grupo de consultas inteiro for considerado, é possível descobrir subexpressões comuns que possam ser avaliadas uma vez para o grupo inteiro. Finkelstein [1982] e Hall [1976] consideram a otimização de um grupo de consultas e o uso de subexpressões comuns. Dalvi *et al.* [2001] discutem questões de otimização na canalização com espaço de buffer limitado com compartilhamento de subexpressões comuns.

Otimização da consulta semântica: a otimização da consulta pode utilizar a informação de semântica, como dependências funcionais e outras restrições de integridade. A *otimização da consulta semântica* nos bancos de dados relacionais é explicada por King [1981], Chakravarthy *et al.* [1990] e, no contexto da agregação, por Sudarshan e Ramakrishnan [1991].

Views materializadas: Blakeley *et al.* [1986], Blakeley *et al.* [1986, 1989] e Griffin e Libkin [1995] descrevem técnicas para a manutenção de views materializadas. Gupta e Mumick [1995] oferecem um levantamento da manutenção de views materializadas. A otimização dos planos de manutenção de view materializada é descrita por Vista [1998] e Mistry *et al.* [2001]. A otimização da consulta na presença de views materializadas é tratada por Larson e Yang [1985], Chaudhuri e outros [1995], Dar *et al.* [1996] e Roy *et al.* [2000]. A seleção de índice e a seleção de view materializada são tratadas por Ross *et al.* [1996], Labio *et al.* [1997], Gupta [1997], Chaudhuri e Narasayya [1997] e Roy e outros [2000]. Aspectos de seleção de índice e view materializada são discutidos mais adiante nas seções "Ajuste de desempenho", "Usando visões materializadas" e "Ajuste automatizado do projeto físico" no Capítulo 23.

Otimização de atualizações: Galindo-Legaria *et al.* [2004] descrevem o processamento e a otimização de consulta para atualizações de banco de dados, incluindo otimização de manutenção de índice, planos de manutenção de view materializada e verificação de restrição de integridade. Consultas de atualização normalmente envolvem subconsultas nas cláusulas **set** e **where**, que também precisam ser levadas em conta na otimização da atualização.

As atualizações que envolvem uma seleção sobre a coluna atualizada (por exemplo, dê um aumento de salário de 10% a todos os funcionários cujo salário é ≥ 100.000) precisam ser tratadas cuidadosamente. Se a atualização é feita enquanto a seleção está sendo avaliada por uma varredura de índice, uma tupla atualizada pode ser reinserida no índice antes da varredura e vista novamente pela varredura; a mesma tupla de funcionário pode, então, ser atualizada incorretamente várias vezes (um número infinito de vezes, nesse caso). Um problema semelhante também surge com atualizações envolvendo subconsultas cujo resultado é afetado pela atualização.

O problema de uma atualização afetando a execução de uma consulta associada à atualização é conhecido como *problema do Halloween* (devido à data em que foi reconhecido inicialmente na IBM). O problema pode ser evitado pela

execução de consultas definindo a atualização primeiro, criando uma lista de tuplas afetadas e atualizando as tuplas e índices como a última etapa. Porém, dividir o plano de execução dessa maneira aumenta o custo da execução. Os planos de atualização podem ser otimizados verificando se o problema do Halloween pode ocorrer e, se não puder, as atualizações podem ser realizadas enquanto a consulta está sendo processada, reduzindo as sobrecargas da atualização. Essa otimização é implementada na maioria dos sistemas de banco de dados e é descrita, por exemplo, em Galindo-Legaria *et al.* [2004].

Indexação XML, processamento e otimização de consulta: A indexação de dados XML e o processamento e otimização de consultas XML tem sido uma área de muito interesse nos últimos anos. Diversos artigos foram publicados sobre o assunto. Um dos desafios na indexação é que as consultas podem especificar uma seleção sobre um caminho, como */a/b//c[d="CSE"]*; o índice precisa dar suporte à recuperação eficiente dos nós que satisfazem a especificação de caminho e a seleção de valor. Trabalhos recentes sobre a indexação de dados XML incluem Pal *et al.* [2004] e Kaushik et al. [2004]. Se os dados forem desmembrados e armazenados em relações, a avaliação de uma expressão de caminho é mapeada para o cálculo de uma junção. Várias técnicas foram propostas para calcular de forma eficiente essas junções, em particular, quando a expressão de caminho especifica qualquer descendente (*//*). Várias técnicas para numeração de nós em dados XML foram propostas, podendo ser usadas para verificar de forma eficiente se um nó é um descendente de outro; veja, por exemplo, O'Neil *et al.* [2004]. O trabalho sobre a otimização de consultas XML inclui McHugh e Widom [1999], Wu *et al.* [2003] e Krishnaprasad *et al.* [2004].

PARTE 5

Gerenciamento de transação

O termo *transação* refere-se a uma coleção de operações que formam uma única unidade de trabalho lógica. Por exemplo, a transferência de dinheiro de uma conta para outra é uma transação consistindo em duas atualizações, uma para cada conta.

É importante que todas as ações de uma transação sejam executadas completamente ou, no caso de alguma falha, efeitos parciais de cada transação incompleta sejam desfeitos. Essa propriedade é chamada de *atomicidade*. Além disso, quando uma transação é executada com sucesso, seus efeitos precisam persistir no banco de dados – uma falha no sistema não deverá resultar no banco de dados se esquecendo de uma transação que foi completada com sucesso. Essa propriedade é chamada de *durabilidade*.

Em um sistema de banco de dados em que várias transações estão executando ao mesmo tempo, se as atualizações nos dados compartilhados não forem controladas, haverá potencial para transações verem estados intermediários inconsistentes, criados por atualizações de outras transações. Essa situação pode resultar em atualizações erradas aos dados armazenados no banco de dados. Assim, os sistemas de banco de dados precisam oferecer mecanismos para isolar transações dos efeitos das outras transações executando simultaneamente. Essa propriedade é chamada de *isolamento*.

O Capítulo 15 descreve o conceito de uma transação com detalhes, incluindo as propriedades de atomicidade, durabilidade, isolamento e outras oferecidas pela abstração da transação. Em particular, o capítulo esclarece a noção de isolamento, por meio de um conceito chamado seriação.

O Capítulo 16 descreve várias técnicas de controle de concorrência que ajudam a implementar a propriedade de isolamento. O Capítulo 17 descreve o componente de gerenciamento de recuperação de um banco de dados, que implementa as propriedades de atomicidade e durabilidade.

Tomando como um todo, o componente de gerenciamento de transação de um sistema de banco de dados permite que os desenvolvedores de aplicação focalizem a implementação de transações individuais, ignorando as questões de concorrência e tolerância a falhas.

CAPÍTULO 15

Transações

Normalmente, uma coleção de várias operações sobre o banco de dados parece ser uma única entidade do ponto de vista do usuário do banco de dados. Por exemplo, uma transferência de fundos de uma conta-corrente para uma conta de poupança é uma única operação do ponto de vista do cliente; contudo, dentro do sistema de banco de dados, ela consiste em várias operações. Logicamente, é essencial que todas essas operações ocorram ou que, no caso de uma falha, nenhuma delas ocorra. Seria inaceitável se a conta-corrente fosse debitada, mas a conta de poupança não fosse creditada.

Coleções de operações que formam uma única unidade lógica de trabalho são chamadas **transações**. Um sistema de banco de dados precisa garantir a execução apropriada de transações apesar das falhas – ou a transação inteira é executada ou nenhuma parte dela é executada. Além do mais, ele precisa gerenciar a execução simultânea de transações de um modo a evitar a introdução da inconsistência. Em nosso exemplo de transferência de fundos, uma transação calculando o valor total do cliente poderia ver o saldo da conta-corrente antes de ser debitado pela transação de transferência de fundos, mas ver o saldo da poupança depois de ser creditado. Como resultado, ela obteria um resultado incorreto.

Este capítulo apresenta os conceitos básicos de processamento de transação. Os detalhes sobre processamento de transação e recuperação de falhas estão nos Capítulos 16 e 17, respectivamente. Outros tópicos sobre processamento de transação são discutidos no Capítulo 25.

Conceito de transação

Uma transação é uma unidade de execução do programa que acessa e possivelmente atualiza vários itens de dados. Normalmente, uma transação é iniciada por um programa do usuário escrito em uma linguagem de manipulação de dados ou linguagem de programação de alto nível (por exemplo, SQL, C++ ou Java), em que é delimitada pelas instruções (ou chamadas de função) na forma **begin transaction** e **end transaction**. A transação consiste em todas as operações executadas entre o **begin transaction** e o **end transaction**.

Para garantir a integridade dos dados, é necessário que o sistema de banco de dados mantenha as seguintes propriedades das transações:

- **Atomicidade.** Todas as operações da transação são refletidas corretamente no banco de dados, ou nenhuma delas.
- **Consistência.** A execução de uma transação isolada (ou seja, sem qualquer outra transação executando simultaneamente) preserva a consistência do banco de dados.
- **Isolamento.** Embora várias transações possam ser executadas simultaneamente, o sistema garante que, para cada par de transações T_i e T_j, parece para T_i que ou T_j terminou a execução antes que T_i começasse ou T_j iniciou a execução depois que T_i terminou. Assim, cada transação não está ciente das outras transações executando simultaneamente no sistema.
- **Durabilidade.** Depois que uma transação for completada com sucesso, as mudanças que ela fez ao banco de dados persistem, mesmo que existam falhas no sistema.

Essas propriedades normalmente são conhecidas como **propriedades ACID**; o acrônimo é derivado da primeira letra de cada uma das quatro propriedades.

Para entender melhor as propriedades ACID e a necessidade delas, considere um sistema bancário simplificado,

consistindo em várias contas e um conjunto de transações que acessam e atualizam essas contas. Por enquanto, vamos supor que o banco de dados resida permanentemente no disco, mas que alguma parte dele esteja residindo temporariamente na memória principal.

As transações acessam dados usando duas operações:

- `read`(X), que transfere o item de dados X do banco de dados para um buffer local pertencente à transação que executou a operação `read`.
- `write`(X), que transfere o item de dados X do buffer local da transação que executou o `write` de volta ao banco de dados.

Em um sistema de banco de dados real, a operação `write` não necessariamente resulta na atualização imediata dos dados no disco; a operação `write` pode ser armazenada temporariamente na memória e executada no disco mais tarde. Por enquanto, porém, vamos supor que a operação `write` atualize o banco de dados imediatamente. Retornaremos a esse assunto no Capítulo 17.

Suponha que T_1 seja uma transação que transfere $50 da conta A para a conta B. Essa transação pode ser definida como

```
T1: read(A);
    A := A - 50;
    write(A);
    read(B);
    B := B + 50;
    write(B).
```

Agora, vamos considerar cada uma das propriedades ACID. (Para facilitar a apresentação, consideramos as propriedades em uma ordem diferente da ordem A–C–I–D.)

- **Consistência**: o requisito de consistência aqui é que a soma de A e B sejam inalteradas pela execução da transação. Sem o requisito de consistência, o dinheiro poderia ser criado ou destruído pela transação! Podemos verificar facilmente que, se o banco de dados estiver consistente antes de uma execução da transação, ele permanecerá consistente após a execução da transação.

  Garantir a consistência para uma transação individual é responsabilidade do programador de aplicação que codifica a transação. Essa tarefa pode ser facilitada pelo teste automático de restrições de integridade, conforme discutimos na seção "Restrições de integridade" do Capítulo 4.
- **Atomicidade**: suponha que, imediatamente antes da execução da transação T_1, os valores das contas A e B sejam $1.000 e $2.000, respectivamente. Agora suponha que, durante a execução da transação T_1, aconteça uma falha que impede T_1 de completar sua execução com sucesso. Os exemplos dessas falhas incluem falta de energia, falhas de hardware e erros de software. Além disso, suponha que a falha aconteceu depois da operação `write`(A), mas antes da operação `write`(B). Nesse caso, os valores das contas A e B refletidos no banco de dados são $950 e $2.000. O sistema destruiu $50 como resultado dessa falha. Em particular, notamos que a soma $A + B$ não é mais preservada.

  Assim, devido à falha, o estado do sistema não reflete mais um estado real do mundo que o banco de dados deveria capturar. Chamamos esse estado de **estado inconsistente**. Temos de garantir que essas inconsistências não sejam visíveis em um sistema de banco de dados. Observe, porém, que o sistema precisa, em algum ponto, estar em um estado inconsistente. Mesmo que a transação T_1 seja executada até o término, existe um ponto em que o valor da conta A é $950 e o valor da conta B é $2.000, o que claramente é um estado inconsistente. Porém, esse estado por fim será substituído pelo estado consistente, em que o valor da conta A é $950 e o valor da conta B é $2.050. Assim, se a transação nunca foi iniciada ou foi completada com sucesso, esse estado inconsistente não seria visível, exceto durante a execução da transação. Ou seja, o motivo para o requisito de atomicidade: se a propriedade de atomicidade estiver presente, todas as ações da transação são refletidas no banco de dados, ou nenhuma delas.

  A idéia básica por trás da garantia da atomicidade é esta: o sistema de banco de dados acompanha (em disco) os valores antigos de quaisquer dados em que uma transação realiza uma escrita e, se a transação não completar sua execução, o sistema de banco de dados restaura os valores antigos para que apareçam como se a transação nunca tivesse sido executada. Discutimos melhor essas idéias na próxima seção. Garantir a atomicidade é responsabilidade do próprio sistema de banco de dados; especificamente, ela é tratada por um componente chamado **componente de gerenciamento de transação**, que descrevemos com detalhes no Capítulo 17.
- **Durabilidade**: quando a execução da transação termina com sucesso, e o usuário que iniciou a transação foi notificado de que a transferência de fundos aconteceu, é preciso que nenhuma falha no sistema resulte em uma perda dos dados correspondentes a essa transferência de fundos.

  A propriedade de durabilidade garante que, quando uma transação é executada com sucesso, todas as atualizações que ela executou no banco de dados persistem, mesmo que haja uma falha no sistema após a transação terminar sua execução.

  Consideramos por enquanto que uma falha do sistema de computador pode resultar na perda de dados na

memória principal, mas os dados gravados em disco nunca são perdidos. Podemos garantir a durabilidade garantindo que

1. As atualizações executadas pela transação sejam gravadas em disco antes que a transação termine.
2. As informações sobre as atualizações executadas pela transação e gravadas em disco sejam suficientes para permitir que o banco de dados reconstrua as atualizações quando o sistema de banco de dados for reiniciado após a falha.

Garantir a durabilidade é responsabilidade de um componente de software do sistema de banco de dados, chamado **componente de gerenciamento de recuperação**. O componente de gerenciamento de transação e o componente de gerenciamento de recuperação estão bastante relacionados, e são descritos no Capítulo 17.

- **Isolamento**: mesmo que as propriedades de consistência e atomicidade sejam garantidas para cada transação, se várias transações forem executadas simultaneamente, suas operações podem intercalar de alguma maneira indesejável, resultando em um estado inconsistente.

Por exemplo, como já vimos, o banco de dados está temporariamente inconsistente enquanto a transação para transferir fundos de A para B está executando, com o total deduzido gravado em A e o total aumentado ainda a ser gravado em B. Se uma segunda transação executando simultaneamente lê A e B nesse ponto intermediário e calcula $A + B$, ela observará um valor inconsistente. Além do mais, se essa segunda transação realizar atualizações sobre A e B com base nos valores inconsistentes que ela lê, o banco de dados pode ser deixado em um estado inconsistente mesmo depois que as transações tenham terminado.

Um modo de evitar o problema de transações executando simultaneamente é executar transações de modo serial – ou seja, uma após a outra. Porém, a execução simultânea de transações oferece benefícios de desempenho significativos, como veremos na seção "Execuções simultâneas". Portanto, outras soluções foram desenvolvidas; elas permitem que várias transações sejam executadas simultaneamente.

Discutimos os problemas causados pelas transações executando simultaneamente na seção "Execuções simultâneas". A propriedade de isolamento de uma transação garante que a execução simultânea das transações resulte em um estado do sistema equivalente ao estado que poderia ter sido obtido se essas transações fossem executadas uma de cada vez em alguma ordem. Discutiremos os princípios do isolamento mais adiante, na seção "Seriação". A garantia da propriedade de isolamento é responsabilidade de um componente do sistema de banco de dados chamado **componente de controle de concorrência**, que discutimos mais adiante, no Capítulo 16.

Estado da transação

Na ausência de falhas, todas as transações são completadas com sucesso. Porém, como observamos anteriormente, uma transação nem sempre pode completar sua execução com sucesso. Essa transação é considerada **abortada**. Se tivermos de garantir a propriedade de atomicidade, uma transação abordada não pode ter efeito sobre o estado do banco de dados. Assim, quaisquer mudanças que a transação abordada fez no banco de dados precisam ser desfeitas. Quando as mudanças causadas por uma transação abortada tiverem sido desfeitas, dizemos que a transação foi **revertida** (*rolled back*). É parte da responsabilidade do esquema de recuperação gerenciar transações revertidas.

Uma transação que completa sua execução com sucesso é considerada **confirmada** (*committed*). Uma transação confirmada que realizou atualizações transforma o banco de dados em um novo estado consistente, que precisa persistir mesmo que haja uma falha no sistema.

Quando uma transação tiver sido confirmada, não podemos desfazer seus efeitos abortando-a. A única maneira de desfazer os efeitos de uma transação confirmada é executar uma **transação de compensação**. Por exemplo, se uma transação somasse $20 a uma conta, a transação de compensação subtrairia $20 da conta. Porém, nem sempre é possível criar essa transação de compensação. Portanto, a responsabilidade de escrever e executar uma transação de compensação fica para o usuário, e não é tratada pelo sistema de banco de dados. O Capítulo 25 inclui uma discussão sobre transações de compensação.

Precisamos ser mais precisos a respeito do que significa *término bem-sucedido* de uma transação. Portanto, vamos estabelecer um modelo abstrato simples de transação. Uma transação precisa estar em um dos seguintes estados:

- **Ativa**, o estado inicial; a transação permanece nesse estado enquanto está executando
- **Parcialmente confirmada**, depois que a instrução final foi executada
- **Falha**, depois da descoberta de que a execução normal não pode mais prosseguir
- **Abortada**, depois que a transação foi revertida e o banco de dados foi restaurado ao seu estado anterior ao início da transação
- **Confirmada**, após o término bem-sucedido

O diagrama de estado corresponde a uma transação aparece na Figura 15.1. Dizemos que uma transação foi confir-

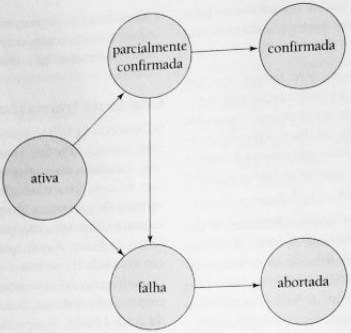


Figura 15.1 Diagrama de estado de uma transação.

mada somente se ela entrou no estado confirmado. De modo semelhante, dizemos que uma transação foi abortada somente se ela entrou no estado abortado. Uma transação é considerada como **terminada** se tiver sido confirmada ou abortada.

Uma transação começa no estado ativo. Quando ela termina sua última instrução, entra no estado parcialmente confirmado. Nesse ponto, a transação completou sua execução mas ainda pode ser abortada, pois a saída real ainda pode estar temporariamente residindo na memória principal, e por isso uma falha de hardware pode impedir seu término bem-sucedido.

O sistema de banco de dados, então, escreve informações suficientes em disco que, mesmo no caso de uma falha, as atualizações realizadas pela transação possam ser recriadas quando o sistema reiniciar após uma falha. Quando a última informação for gravada, a transação entra no estado confirmado.

Como já dissemos, consideramos por enquanto que as falhas não resultam em perda de dados no disco. O Capítulo 17 discute as técnicas para lidar com a perda de dados no disco.

Uma transação entra no estado de falha depois que o sistema determina que a transação não pode mais prosseguir com sua execução normal (por exemplo, devido a erros de hardware ou lógicos). Tal transação precisa ser revertida. Depois, ela entra no estado abortado. Nesse ponto, o sistema tem duas opções:

- Ele pode **reiniciar a transação**, mas somente se esta tiver sido abortada como resultado de algum erro de hardware ou software que não foi criado por meio da lógica interna da transação. Uma transação reiniciada é considerada como sendo uma nova transação.
- Ele pode **matar** a transação. Ele normalmente faz isso devido a algum erro lógico interno que só pode ser corrigido com a reescrita do programa de aplicação, ou porque a entrada foi defeituosa, ou porque os dados desejados não foram encontrados no banco de dados.

Precisamos ter cuidado ao tratar de **escritas externas observáveis**, como escritas em um terminal ou impressora. Quando uma escrita tiver ocorrido, ela não poderá ser apagada, pois pode ter sido vista externamente ao sistema de banco de dados. A maioria dos sistemas permite que essas escritas ocorram apenas depois que a transação tiver entrado no estado confirmado. Um modo de implementar esse esquema é para o sistema de banco de dados armazenar qualquer valor associado a escritas externas temporariamente no armazenamento não volátil, e realizar as escritas reais apenas depois que a transação entrar no estado confirmado. Se o sistema falhar após a transação ter entrado no estado confirmado, mas antes que pudesse completar as escritas externas, o sistema de banco de dados executará as escritas externas (usando os dados no armazenamento não volátil) quando o sistema for reiniciado.

O tratamento de escritas externas pode ser mais complicado em algumas situações. Por exemplo, suponha que a ação externa seja a de expedir dinheiro em um caixa eletrônico, e o sistema falhe imediatamente antes que o dinheiro seja realmente entregue (supomos que o dinheiro possa ser entregue atomicamente). Não faz sentido entregar o dinhei-

ro quando o sistema for reiniciado, pois o usuário pode ter saído da máquina. Nesse caso, uma transação de compensação, como depositar o dinheiro de volta na conta do usuário, precisa ser executada quando o sistema for reiniciado.

Para certas aplicações, pode ser desejável permitir que transações ativas apresentem dados aos usuários, particularmente para transações de longa duração que rodam por minutos ou horas. Infelizmente, não podemos permitir tal saída de dados observáveis sem comprometimento da atomicidade da transação. A maioria dos sistemas de transação atuais garante a atomicidade e, portanto, proíbe essa forma de interação com usuários. No Capítulo 25, discutimos os modelos de transação alternativos, que dão suporte a transações interativas, de longa duração.

Implementação de atomicidade e durabilidade

O componente de gerenciamento de recuperação de um sistema de banco de dados pode dar suporte a atomicidade e durabilidade por uma série de esquemas. Primeiro, consideramos um esquema simples, porém extremamente ineficaz, chamado esquema de **cópia de sombra**. Esse esquema, que é baseado em fazer cópias do banco de dados, chamadas cópias de *sombra*, considera que somente uma transação está ativa ao mesmo tempo. O esquema também considera que o banco de dados é simplesmente um arquivo no disco. Um ponteiro, chamado db-pointer, é mantido no disco; ele aponta para a cópia atual do banco de dados.

No esquema de cópia de sombra, uma transação que quer atualizar o banco de dados primeiro cria uma cópia completa dele. Todas as atualizações são feitas sobre a nova cópia do banco de dados, deixando a cópia original, a **cópia de sombra**, intocável. Se, a qualquer momento, a transação tiver de ser abortada, o sistema simplesmente exclui a nova cópia. A cópia antiga do banco de dados não foi afetada.

Se a transação for concluída, ela será confirmada da seguinte maneira. Primeiro, o sistema operacional é solicitado a confirmar se todas as páginas da nova cópia do banco de dados foram gravadas em disco. (Os sistemas Unix usam o comando fsync para essa finalidade.) Depois que o sistema operacional tiver gravado todas as páginas em disco, o sistema de banco de dados atualiza o ponteiro db-pointer para que aponte para a nova cópia do banco de dados; a nova cópia, então, torna-se a cópia atual do banco de dados. A cópia antiga do banco de dados é, então, excluída. A Figura 15.2 representa o esquema, mostrando o estado do banco de dados antes e depois da atualização. Diz-se que a transação foi *confirmada* no ponto em que o db-pointer atualizado é gravado no disco.

Agora, vamos considerar como a técnica trata de falhas da transação e do sistema. Primeiro, considere a falha da transação. Se a transação falhar no momento antes que o db-pointer é atualizado, o conteúdo antigo do banco de dados não é afetado. Podemos abortar a transação simplesmente excluindo a nova cópia do banco de dados. Quando a transação tiver sido confirmada, todas as atualizações que ela realizou estão no banco de dados apontado por db-pointer. Assim, ou todas as atualizações da transação são refletidas, ou nenhum dos efeitos é refletido, independente da falha da transação.

Agora, considere a questão da falha do sistema. Suponha que o sistema falhe em qualquer momento antes que o db-pointer atualizado seja gravado em disco. Então, quando o sistema for reiniciado, ele lerá o db-pointer e, assim, verá o conteúdo original do banco de dados, e nenhum dos efeitos da transação será visível no banco de dados. Em seguida, suponha que o sistema falhe depois que o db-pointer tiver sido atualizado em disco. Antes que o db-pointer fosse atualizado, todas as páginas atualizadas da nova cópia do banco de dados foram gravadas em disco. Novamente, supomos que, quando um arquivo é gravado em disco, seu conteúdo não

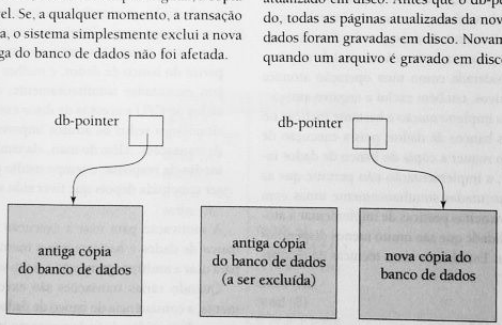


Figura 15.2 Técnica de cópia de sombra para atomicidade e durabilidade.

será danificado mesmo que haja uma falha no sistema. Portanto, quando o sistema reiniciar, ele lerá o db-pointer e, assim, verá o conteúdo do banco de dados *depois* de todas as atualizações serem realizadas pela transação.

A implementação realmente depende de a escrita no db-pointer ser atômica; ou seja, ou todos os seus bytes são gravados ou nenhum de seus bytes é gravado. Se alguns dos bytes do ponteiro foram atualizados pela escrita, mas outros não, o ponteiro não terá significado, e nem as versões antigas nem as novas do banco de dados poderão ser encontradas quando o sistema reiniciar. Felizmente, os sistemas de disco oferecem atualizações atômicas a blocos inteiros, ou, pelo menos, a um setor do disco. Em outras palavras, o sistema de disco garante que atualizará o db-pointer atomicamente, desde que estejamos certos de que o db-pointer se encontra inteiramente em um único setor, o que podemos garantir classificando o db-pointer no início de um bloco.

Assim, as propriedades de atomicidade e durabilidade das transações são garantidas pela implementação de cópia de sombra do componente de gerenciamento de recuperação.

Como exemplo simples de uma transação fora do domínio do banco de dados, considere uma sessão de edição de textos. Uma sessão de edição de textos pode ser modelada como uma transação. As ações executadas pela transação são leitura e atualização do arquivo. Salvar o arquivo no final da edição corresponde a um commit da transação de edição; sair da sessão de edição sem salvar o arquivo corresponde a abortar a transação de edição.

Muitos editores de textos utilizam basicamente a implementação recém-descrita, para garantir que uma sessão de edição seja transacional. Um novo arquivo é usado para armazenar o arquivo atualizado. Ao final da sessão de edição, se o arquivo atualizado tiver de ser salvo, o editor de textos usa um comando de *renomeação* de arquivo, a fim de renomear o novo arquivo para que tenha o nome real do arquivo. A renomeação, considerada como uma operação atômica pelo sistema de arquivos, também exclui o arquivo antigo.

Infelizmente, essa implementação é bastante ineficaz no contexto de grandes bancos de dados, pois a execução de uma única transação requer a cópia do banco de dados *inteiro*. Além do mais, a implementação não permite que as transações sejam executadas simultaneamente umas com as outras. Existem maneiras práticas de implementar a atomicidade e a durabilidade que são muito menos dispendiosas e mais poderosas. Estudamos essas técnicas de recuperação no Capítulo 17.

Execuções simultâneas

Os sistemas de processamento de transação normalmente permitem que várias transações sejam executadas simultaneamente. Permitir que várias transações atualizem dados simultaneamente causa várias complicações com consistência de dados, como vimos anteriormente. Garantir a consistência apesar da execução simultânea de transações exige trabalho extra; é muito mais fácil insistir que as transações sejam executadas **serialmente** – ou seja, uma de cada vez, cada uma começando apenas depois que a anterior for concluída. Porém, existem dois bons motivos para permitir a concorrência:

- **Melhor throughput e utilização de recursos.** Uma transação consiste em muitas etapas. Algumas envolvem atividades de E/S; outras envolvem atividade de CPU. A CPU e os discos em um sistema de computador podem operar em paralelo. Portanto, a atividade de E/S pode ser feita em paralelo com o processamento na CPU. O paralelismo da CPU e do sistema de E/S pode, portanto, ser explorado para executar várias transações em paralelo. Enquanto um `read` ou `write` em favor de uma transação está em andamento em um disco, outra transação pode estar executando na CPU, enquanto outro disco pode estar executando um `read` ou `write` em favor de uma terceira transação. Tudo isso aumenta o **throughput** do sistema – ou seja, o número de transações executadas em determinada quantidade de tempo. De modo correspondente, a **utilização** do processador e do disco também aumenta; em outras palavras, o processador e o disco passam menos tempo ociosos, ou não realizando algum trabalho útil.
- **Tempo de espera reduzido.** Pode haver uma mistura de transações executando em um sistema, algumas curtas e outras longas. Se as transações forem executadas serialmente, uma transação curta pode ter de esperar que uma transação longa anterior seja concluída, o que pode levar a atrasos imprevisíveis na execução de uma transação. Se as transações estão operando sobre diferentes partes do banco de dados, é melhor deixar que elas sejam executadas simultaneamente, compartilhando os ciclos de CPU e acessos de disco entre elas. A execução simultânea reduz os atrasos imprevisíveis na execução de transações. Além do mais, ela também reduz o **tempo médio de resposta**: o tempo médio para uma transação ser concluída depois que tiver sido submetida.

A motivação para usar a execução simultânea em um banco de dados é basicamente a mesma que a motivação para usar a **multiprogramação** em um sistema operacional.

Quando várias transações são executadas simultaneamente, a consistência do banco de dados pode ser destruída apesar da exatidão de cada transação individual. Nesta seção, apresentamos o conceito de schedules para ajudar a identificar as execuções que garantem a consistência.

O sistema de banco de dados precisa controlar a interação entre as transações simultâneas para impedir que destruam a consistência do banco de dados. Ele faz isso por meio de uma série de mecanismos, chamados **esquemas de controle de consistência**. Estudamos os esquemas de controle de concorrência no Capítulo 16; por enquanto, focalizamos o conceito da execução simultânea correta.

Considere novamente o sistema bancário simplificado da primeira seção deste capítulo, que possui várias contas, e um conjunto de transações que acessam e atualizam essas contas. Suponha que T_1 e T_2 sejam duas transações que transferem fundos de uma conta para outra. A transação T_1 transfere $50 da conta A para a conta B. Ela é definida como

```
T1: read(A);
    A := A - 50;
    write(A);
    read(B);
    B := B + 50;
    write(B).
```

A transação T_2 transfere 10% do saldo da conta A para a conta B. Ela é definida como

```
T2: read(A);
    temp := A * 0.1;
    A := A - temp;
    write(A);
    read(B);
    B := B + temp;
    write(B).
```

Suponha que os valores atuais das contas A e B sejam $1.000 e $2.000, respectivamente. Suponha também que as duas transações sejam executadas uma de cada vez na ordem T_1 seguida por T_2. Essa seqüência de execução aparece na Figura 15.3. Na figura, a seqüência de etapas de instrução está em ordem cronológica de cima para baixo, com instruções de T_1 aparecendo na coluna esquerda e as instruções de T_2 aparecendo na coluna da direita. Os valores finais das contas A e B, após a execução na Figura 15.3, são $855 e $2.145, respectivamente. Assim, a quantia total de dinheiro nas contas A e B – ou seja, a soma $A + B$ – é preservada depois da execução das duas transações.

De modo semelhante, se as transações forem executadas uma de cada vez na ordem T_2 seguida por T_1, então a seqüência de execução é a da Figura 15.4. Novamente, conforme esperamos, a soma $A + B$ é preservada, e os valores finais das contas A e B são $850 e $2.150, respectivamente.

As seqüências de execução recém-descritas são chamadas **schedules**. Elas representam a ordem cronológica em que as instruções são executadas no sistema. Claramente, um schedule para um conjunto de transações precisa consistir em todas as instruções dessas transações, e precisa preservar a ordem em que as instruções aparecem em cada transação individual. Por exemplo, na transação T_1, a instrução `write(A)` precisa aparecer antes da instrução `read(B)`, em qualquer schedule válido. Na discussão a seguir, vamos nos referir à primeira seqüência de execução (T_1 seguida por T_2) como schedule 1, e à segunda seqüência de execução (T_2 seguida por T_1) como schedule 2.

Esses schedules são **seriais**: cada schedule serial consiste em uma seqüência de instruções de várias transações, em que as instruções pertencentes a uma única transação aparecem juntas nesse schedule. Assim, para um conjunto de n transações, existem $n!$ diferentes schedules seriais válidos.

Quando o sistema de banco de dados executa várias transações simultaneamente, o schedule correspondente não precisa mais ser serial. Se duas transações estão executando simultaneamente, o sistema operacional pode executar uma transação por um tempo, depois realizar uma troca de contexto, executar a segunda transação por algum tempo e depois voltar à primeira transação por algum tempo, e

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write (B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)

Figura 15.3 Shedule 1 – um schedule serial em que T_1 é seguido por T_2.

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read(B)
	$B := B + temp$
	write (B)
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write (B)	

Figura 15.4 Schedule 2 – um schedule serial em que T_2 é seguida por T_1.

assim por diante. Com várias transações, o tempo de CPU é compartilhado entre todas as transações.

Várias seqüências de execução são possíveis, pois as várias instruções das duas transações agora podem ser intercaladas. Em geral, não é possível prever exatamente quantas instruções de uma transação serão executadas antes que a CPU passe para outra transação. Assim, o número de schedules possíveis para um conjunto de n transações é muito maior que $n!$.

Retornando ao nosso exemplo anterior, suponha que as duas transações sejam executadas simultaneamente. Um schedule possível aparece na Figura 15.5. Depois que essa execução acontece, chegamos ao mesmo estado daquele em que as transações são executadas em série na ordem T_1 seguida por T_2. A soma $A + B$ é realmente preservada.

Nem todas as execuções concorrentes resultam em um estado correto. Para ilustrar, considere o schedule da Figura 15.6. Depois da execução desse schedule, chegamos a um estado em que os valores finais das contas A e B são $950 e $2.100, respectivamente. Esse estado final é um *estado inconsistente*, pois ganhamos $50 no processo da execução simultânea. Na realidade, a soma $A + B$ não é preservada pela execução das duas transações.

Se o controle da execução concorrente for deixado inteiramente para o sistema operacional, muitos schedules possíveis – incluindo aqueles que deixam o banco de dados em um estado inconsistente, como aquele que acabamos de descrever – são possíveis. É tarefa do sistema de banco de dados garantir que qualquer schedule executado deixe o banco de dados em um estado consistente. O **componente de controle de concorrência** do sistema de banco de dados executa sua tarefa.

Podemos garantir a consistência do banco de dados sob execução simultânea cuidando para que qualquer schedule executado tenha o mesmo efeito do schedule que poderia ter ocorrido sem qualquer execução simultânea. Ou seja, o schedule precisa, de certa forma, ser equivalente a um schedule serial. Examinamos essa idéia a seguir.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Figura 15.5 Schedule 3 – um schedule simultâneo equivalente ao schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Figura 15.6 Schedule 4 – um schedule simultâneo.

Seriação

O sistema de banco de dados precisa controlar a execução simultânea de transações, para garantir que o estado do banco de dados permaneça consistente. Antes de examinar como o sistema de banco de dados pode executar essa tarefa, primeiro temos de entender quais schedules garantirão a consistência e quais não garantirão.

Como as transações são programas, é computacionalmente difícil determinar exatamente que operações uma transação realiza e como as operações de várias transações interagem. Por esse motivo, não interpretaremos o tipo de operações que uma transação pode realizar sobre um item de dados. Em vez disso, consideramos apenas duas operações: `read` e `write`. Assim, consideramos que, entre uma instrução `read`(Q) e uma instrução `write`(Q) sobre um item de dados Q, uma transação pode realizar uma seqüência qualquer de operações sobre a cópia de Q que está residindo no buffer local da transação. Assim, as únicas operações significativas de uma transação, a partir de um ponto de vista do escalonamento, são suas instruções `read` e `write`. Portanto, normalmente mostraremos apenas instruções `read` e `write` nos schedules, como fazemos no schedule 3, na Figura 15.7.

Nesta seção, discutimos diferentes formas de equivalência em schedule; elas levam às noções de **seriação de conflito** e **seriação visão**.

Seriação de conflito

Vamos considerar um schedule S em que existem duas instruções consecutivas, I_i e I_j, de transações T_i e T_j, respectivamente (i j). Se I_i e I_j se referem a diferentes itens de dados, então podemos inverter I_i e I_j sem afetar os resultados de qualquer instrução no schedule. Porém, se I_i e I_j se referem ao mesmo item de dados Q, então a ordem das duas etapas pode importar. Como estamos lidando apenas com instruções `read` e `write`, existem quatro casos que precisamos considerar:

1. I_i = `read`(Q), I_j = `read`(Q). A ordem de I_i e I_j não importa, pois o mesmo valor de Q é lido por T_i e T_j, independente da ordem.
2. I_i = `read`(Q), I_j = `write`(Q). Se I_i vem antes de I_j, então T_i não lê o valor de Q que é escrito por T_j na instrução I_j. Se I_j vem antes de I_i, então T_i lê o valor de Q que é escrito por T_j. Assim, a ordem de I_i e I_j importa.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figura 15.7 Schedule 3 – mostrando apenas as instruções `read` e `write`.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Figura 15.8 Schedule 5 – schedule 3 após a troca de um par de instruções.

3. I_i = `write`(Q), I_j = `read`(Q). A ordem de I_i e I_j importa por motivos semelhantes àqueles do caso anterior.
4. I_i = `write`(Q), I_j = `write`(Q). Como as duas instruções são operações `write`, a ordem dessas instruções não afeta T_i ou T_j. Contudo, o valor obtido pela próxima instrução `read`(Q) de S é afetada, pois o resultado apenas da última das duas instruções `write` é preservado no banco de dados. Se não houver outra instrução `write`(Q) após I_i e I_j em S, então a ordem de I_i e I_j afeta diretamente o valor final de Q no estado do banco de dados que resulta do schedule S.

Assim, somente no caso em que I_i e I_j são instruções `read`, a ordem relativa de sua execução não importa.

Dizemos que I_i e I_j **conflitam** se forem operações por diferentes transações sobre o mesmo item de dados, e pelo menos uma dessas operações for uma operação `write`.

Para ilustrar o conceito de instruções conflitantes, consideramos o schedule 3, na Figura 15.7. A instrução `write`(A) de T_1 conflita com a instrução `read`(A) de T_2. Porém, a instrução `write`(A) de T_2 não conflita com a instrução `read`(B) de T_1, pois as duas instruções acessam itens de dados diferentes. Considere que I_i e I_j sejam as instruções consecutivas de um schedule S. Se I_i e I_j forem instruções de diferentes transações e I_i e I_j não conflitarem, então podemos inverter a ordem de I_i e I_j para produzir um novo schedule S'. Esperamos que S seja equivalente a S', pois todas as instruções aparecem na mesma ordem nos dois schedules, exceto por I_i e I_j, cuja ordem não importa.

Como a instrução `write`(A) de T_2 no schedule 3 da Figura 15.7 não conflita com a instrução `read`(B) de T_1, podemos inverter essas instruções para gerar um schedule equivalente, o schedule 5, na Figura 15.8. Independente do estado inicial do sistema, os schedules 3 e 5 produzem o mesmo estado final do sistema. Continuamos a inverter instruções não conflitantes:

- Troque a instrução `read`(B) de T_1 pela instrução `read`(A) de T_2.
- Troque a instrução `write`(B) de T_1 pela instrução `write`(A) de T_2.
- Troque a instrução `write`(B) de T_1 pela instrução `read`(A) de T_2.

O resultado final dessas trocas, o schedule 6 da Figura 15.9, é um schedule serial. Assim, mostramos que o schedule 3 é equivalente a um schedule serial. Essa equivalência implica que, independente do estado inicial do sistema, o schedule 3 produzirá o mesmo estado final que algum schedule serial produzirá.

Se um schedule S puder ser transformado em um schedule S' por uma série de trocas de instruções não conflitantes, dizemos que S e S' são **equivalentes em conflito**.

Em nossos exemplos anteriores, o schedule 1 não é equivalente em conflito ao schedule 2. Porém, o schedule 1

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figura 15.9 Schedule 6 – um schedule serial que é equivalente ao schedule 3.

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Figura 15.10 Schedule 7.

é equivalente em conflito com o schedule 3, pois as instruções `read`(B) e `write`(B) de T_1 podem ser trocadas com as instruções `read`(A) e `write`(A) de T_2.

O conceito de equivalência em conflito leva ao conceito de seriação de conflito. Dizemos que um schedule S é **serial de conflito** se for equivalente em conflito a um schedule serial. Assim, o schedule 3 é serial de conflito, pois é equivalente em conflito ao schedule serial 1.

Finalmente, considere o schedule 7 da Figura 15.10; ele consiste apenas nas operações significativas (ou seja, o `read` e o `write`) das transações T_3 e T_4. Esse schedule não é serial de conflito, pois não é equivalente ao schedule serial $<T_3,T_4>$ ou ao schedule serial $<T_4,T_3>$.

É possível ter dois schedules que produzem o mesmo resultado, mas que não sejam equivalentes em conflito. Por exemplo, considere a transação T_5, que transfere $10 da conta B para a conta A. Considere que o schedule 9 seja definido na Figura 15.11. Sustentamos que o schedule 8 não é equivalente em conflito com o schedule serial $<T_1,T_5>$, pois, no schedule 8, a instrução `write`(B) de T_5 conflita com a instrução `read`(B) de T_1. Assim, não podemos mover todas as instruções de T_1 antes daquelas de T_5 trocando instruções não conflitantes consecutivas. Porém, os valores finais das contas A e B após a execução do schedule 8 ou o schedule serial $<T_1,T_5>$ são iguais – $960 e $2.040, respectivamente.

Podemos ver, por esse exemplo, que existem definições menos rigorosas da equivalência em schedule do que a equivalência em conflito. Para o sistema determinar que o schedule 8 produz o mesmo resultado do schedule serial $<T_1,T_5>$, ele precisa analisar a computação realizada por T_1 e T_5, em vez de apenas as operações `read` e `write`. Em geral, essa análise é difícil de implementar e computacionalmente dispendiosa. Porém, existem outras definições de equivalência em schedule baseadas puramente nas operações `read` e `write`. Vamos considerar uma definição desse tipo na próxima seção.

*Seriação de visão***

Nesta seção, consideramos uma forma de equivalência que é menos rigorosa do que a equivalência em conflito, mas que, como a equivalência em conflito, é baseada apenas nas operações `read` e `write` das transações.

Considere dois schedules S e S', em que o mesmo conjunto de transações participa dos dois schedules. Os schedules S e S' são considerados **equivalentes em visão** se três condições forem atendidas:

1. Para cada item de dados Q, se a transação T_i ler o valor inicial de Q no schedule S, então a transação T_i, no schedule S', também precisa ler o valor inicial de Q.
2. Para cada item de dados Q, se a transação T_i executar `read`(Q) no schedule S, e se esse valor foi produzido por uma operação `write`(Q) executada pela transação T_j, então a operação `read`(Q) da transação

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Figura 15.11 Schedule 8.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Figura 15.12 Schedule 9 – um schedule serial de visão.

T_j, no schedule S', também precisa ler o valor de Q que foi produzido pela mesma operação `write`(Q) da transação T_j.

3. Para cada item de dados Q, a transação (se houver) que realiza a operação `write`(Q) final no schedule S precisa realizar a operação `write`(Q) final no schedule S'.

As condições 1 e 2 garantem que cada transação lê os mesmos valores nos dois schedules e, portanto, realiza a mesma computação. A condição 3, junto com as condições 1 e 2, garante que os dois schedules resultam no mesmo estado final.

Em nossos exemplos anteriores, o schedule 1 não é equivalente em visão ao schedule 2, pois, no schedule 1, o valor da conta A lida pela transação T_2 foi produzido por T_1, enquanto esse caso não é verdadeiro no schedule 2. Porém, o schedule 1 é equivalente em visão ao schedule 3, pois os valores das contas A e B lidas pela transação T_2 foram produzidos por T_1 nos dois schedules.

O conceito de equivalência em visão leva ao conceito de seriação de visão. Dizemos que um schedule S é **serial de visão** se for equivalente em visão a um schedule serial.

Como ilustração, suponha que aumentamos o schedule 7 com a transação T_6 e obtemos o schedule 9 na Figura 15.12. O schedule 9 é serial de visão. Na realidade, ele é equivalente em visão ao schedule serial <T_3, T_4, T_6>, pois a única instrução `read`(Q) lê o valor inicial de Q nos dois schedules e T_6 realiza a escrita final de Q nos dois schedules.

Cada schedule serial de conflito também é serial de visão, mas existem schedules seriais de visão que não são seriais de conflito. Na realidade, o schedule 9 não é serial de conflito, pois cada par de instruções consecutivas conflita e, sendo assim, nenhuma troca de instruções é possível.

Observe que, no schedule 9, as transações T_4 e T_6 realizam operações `write`(Q) sem ter realizado uma operação `read`(Q). As escritas desse tipo são chamadas **escritas cegas**. As escritas cegas aparecem em qualquer schedule serial de visão que não seja serial de conflito.

Facilidade de recuperação

Até aqui, estudamos que os schedules são aceitáveis do ponto de vista da consistência do banco de dados, considerando implicitamente que não existem falhas de transação. Agora, tratamos do efeito das falhas da transação durante a execução simultânea.

Se uma transação T_i falhar, por qualquer motivo, precisamos desfazer o efeito dessa transação para garantir a propriedade de atomicidade da transação. Em um sistema que permite a execução simultânea, também é necessário garantir que qualquer transação T_j que seja dependente de T_i (ou seja, T_j leu dados escritos por T_i) também seja abortada. Para conseguir essa garantia, precisamos substituir restrições sobre o tipo de schedules permitidos no sistema.

Nas duas subseções a seguir, tratamos da questão de quais schedules são aceitáveis do ponto de vista da recuperação da falha da transação. Descrevemos, no Capítulo 16, como garantir que somente tais schedules aceitáveis sejam gerados.

Schedules recuperáveis

Considere o schedule 10 na Figura 15.13, em que T_9 é uma transação que realiza apenas uma instrução: `read`(A). Suponha que o sistema permita que T_9 seja confirmada imediatamente após executar a instrução `read`(A). Assim, T_9 é confirmada antes de T_8. Agora, suponha que T_8 falhe antes de ser confirmada. Como T_9 leu o valor do item de dados A

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Figura 15.13 Schedule 10.

escrito por T_8, temos de abortar T_9 para garantir a atomicidade da transação. Porém, T_9 já foi confirmada e não pode ser abortada. Assim, temos uma situação em que é impossível recuperar-se corretamente da falha de T_8.

O schedule 10, com o commit acontecendo imediatamente após a instrução read(A), é um exemplo de um schedule *não recuperável*, que não deverá ser permitido. A maioria dos sistemas de banco de dados exige que todos os schedules sejam *recuperáveis*. Um **schedule recuperável** é aquele em que, para cada par de transações T_i e T_j tal que T_j leia um item de dados previamente escrito por T_i, a operação commit de T_i apareça antes da operação commit de T_j.

Schedules não em cascata

Mesmo que um schedule seja recuperável, para recuperar-se corretamente da falha de uma transação T_i, podemos ter de reverter várias transações. Essas situações ocorrem se as transações tiverem lido dados escritos por T_i. Como ilustração, considere o schedule parcial da Figura 15.14. A transação T_{10} escreve um valor de A que é lido pela transação T_{11}. A transação T_{11} escreve um valor de A que é lido pela transação T_{12}. Suponha que, nesse ponto, T_{10} falhe. T_{10} precisa ser revertida. Como T_{11} depende de T_{10}, T_{11} precisa ser revertida. Como T_{12} depende de T_{11}, T_{12} precisa ser revertida. Esse fenômeno, em que a falha de uma única transação leva a uma série de rollbacks de transação, é chamada **rollback em cascata**.

O rollback em cascata é indesejável, pois leva a desfazer uma quantidade de trabalho significativa. É preferível restringir os schedules àqueles em que os rollbacks em cascata não podem ocorrer. Esses schedules são denominados *não em cascata*. Formalmente, um **schedule não em cascata** é aquele em que, para cada par de transações T_i e T_j tal que T_j leia um item de dados previamente escrito por T_i, a operação commit de T_i apareça antes da operação read de T_j. É fácil verificar que cada schedule não em cascata também é recuperável.

Implementação do isolamento

Até aqui, vimos quais propriedades um schedule precisa ter se tiver de deixar o banco de dados em um estado consistente e permitir que as falhas da transação sejam tratadas de uma maneira segura. Especificamente, os schedules que são seriais de conflito ou de visão e não em cascata cumprem esses requisitos.

Existem diversos **esquemas de controle de concorrência** que podemos usar para garantir que, mesmo quando várias transações são executadas simultaneamente, somente os schedules aceitáveis sejam gerados, independente de como o sistema operacional compartilha os recursos com o tempo (como tempo de CPU) entre as transações.

Como exemplo trivial de um esquema de controle de concorrência, considere este esquema: uma transação adquire um **bloqueio** sobre o banco de dados inteiro antes de iniciar e libera o bloqueio depois que ela foi confirmada. Enquanto uma transação mantém um bloqueio, nenhuma outra transação tem permissão para adquirir o bloqueio, e portanto todas precisam esperar que o bloqueio seja liberado. Como resultado da política de bloqueio, somente uma transação pode ser executada de uma vez. Portanto, somente schedules seriais são gerados. Estes são colocados em série de forma trivial, e é fácil verificar que eles também são schedules não em cascata.

Um esquema de controle de concorrência como esse leva a um desempenho inferior, pois força as transações a esperarem até que as transações anteriores terminem antes que possam ser iniciadas. Em outras palavras, ele oferece um grau de concorrência mais baixo. Conforme explicamos na seção "Execuções simultâneas", a execução concorrente possui vários benefícios ao desempenho.

O objetivo dos esquemas de controle de concorrência é oferecer um alto grau de concorrência, enquanto garante que todos os schedules que possam ser gerados sejam seriais de conflito ou visão e não em cascata.

Estudamos diversos esquemas de controle de concorrência no Capítulo 16. Os esquemas possuem diferentes opções em termos da quantidade de concorrência que eles permitem e a quantidade de sobrecarga a que eles ficam sujeitos. Alguns deles só permitem que schedules seriais de conflito sejam gerados; outros permitem que sejam gerados certos schedules seriais de visão que não são seriais de conflito.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Figura 15.14 Schedule 11.

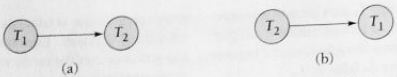


Figura 15.15 Gráfico de precedência para (a) schedule 1 e (b) schedule 2.

Testando a seriação

Ao projetar esquemas de controle de concorrência, temos de mostrar que os schedules gerados pelo esquema são passíveis de seriação. Para isso, primeiro é preciso entender como determinar, dado um schedule S em particular, se o schedule é passível de seriação.

Agora, apresentaremos um método simples e eficiente para determinar a seriação de conflito de um schedule. Considere um schedule S. Construímos um gráfico direcionado, chamado **gráfico de precedência**, a partir de S. Esse gráfico consiste em um par $G = (V, E)$, onde V é um conjunto de vértices e E é um conjunto de arestas. O conjunto de vértices consiste em todas as transações participantes do schedule. O conjunto de arestas consiste em todas as arestas $T_i \rightarrow T_j$ para as quais uma das três condições é verdadeira:

1. T_i executa `write`(Q) antes que T_j execute `read`(Q).
2. T_i executa `read`(Q) antes que T_j execute `write`(Q).
3. T_i executa `write`(Q) antes que T_j `write`(Q).

Se uma aresta $T_i \rightarrow T_j$ existir no gráfico de precedência, então, em qualquer schedule serial S' equivalente a S, T_i precisa aparecer antes de T_j.

Por exemplo, o gráfico de precedência para o schedule 1 na Figura 15.15a contém a única aresta $T_1 \rightarrow T_2$, pois todas as instruções de T_1 são executadas antes que a primeira instrução de T_2 seja executada. Da mesma forma, a Figura 15.15b mostra o gráfico de precedência para o schedule 2 com a única aresta $T_2 \rightarrow T_1$, pois todas as instruções de T_2 são executadas antes de a primeira instrução de T_1 ser executada.

O gráfico de precedência para o schedule 4 aparece na Figura 15.16. Ele contém a aresta $T_1 \rightarrow T_2$, pois T_1 executa `read`(A) antes que T_2 execute `write`(A). Ele também contém a aresta $T_2 \rightarrow T_1$, pois T_2 executa `read`(B) antes que T_1 execute `write`(B).

Se o gráfico de precedência para S tiver um ciclo, então o schedule S não é serial de conflito. Se o gráfico não contém ciclos, então o schedule S é serial de conflito.

Uma **ordem de seriação** das transações pode ser obtida por meio da **classificação topológica**, que determina uma ordem linear consistente com a ordem parcial do gráfico de precedência. Em geral, existem várias ordens lineares possíveis, que podem ser obtidas por meio de uma classificação topológica. Por exemplo, o gráfico da Figura 15.17a possui as duas ordenações lineares aceitáveis mostradas nas Figuras 15.17b e 15.17c.

Assim, para testar a seriação de conflito, precisamos construir o gráfico de precedência e invocar um algoritmo de detecção de ciclo. Os algoritmos de detecção de ciclo podem ser encontrados em livros–texto padrão sobre algoritmos. Os algoritmos de detecção de ciclo, como aqueles baseados na busca primeiro pela profundidade, exigem algo na ordem de n^2 operações, onde n é o número de vértices no gráfico (ou seja, o número de transações). Assim, temos um esquema prático para determinar a seriação de conflito.

Retornando aos nossos exemplos anteriores, observe que os gráficos de precedência para os schedules 1 e 2 (Figura 15.15) na realidade não contêm ciclos. O gráfico de precedência para o schedule 4 (Figura 15.16), por outro lado, contém um ciclo, indicando que esse schedule não é serial de conflito.

O teste da seriação de visão é um tanto complicado. De fato, foi mostrado que o problema de testar a seriação de visão por si só é completamente não procedural. Assim, quase certamente não existe um algoritmo eficiente para testar a seriação de visão.

Veja, nas notas bibliográficas, referências sobre o teste por seriação de visão. Porém, os esquemas de controle de concorrência ainda podem usar *condições suficientes* para a seriação de visão. Ou seja, se condições suficientes forem satisfeitas, o schedule será serial de visão, mas pode haver schedules seriais de visão que não satisfazem as condições suficientes.

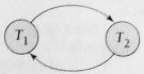


Figura 15.16 Gráfico de precedência para o schedule 4.

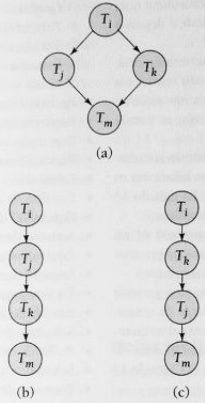


Figura 15.17 Ilustração da classificação topológica.

Resumo

- Uma *transação* é uma unidade de execução de programa que acessa e possivelmente atualiza vários itens de dados. Entender o conceito de uma transação é fundamental para entender e implementar atualizações de dados em um banco de dados de modo que execuções simultâneas e falhas de diversos tipos não resultem na inconsistência do banco de dados.
- As transações precisam ter as propriedades ACID: atomicidade, consistência, isolamento e durabilidade.
  - A atomicidade garante que ou todos os efeitos de uma transação sejam refletidos no banco de dados ou nenhum deles seja aceito; uma falha não pode deixar o banco de dados em um estado em que uma transação é parcialmente executada.
  - A consistência garante que, se o banco de dados for inicialmente consistente, a execução da transação (por si só) deixa o banco de dados em um estado consistente.
  - O isolamento garante que transações executando simultaneamente sejam isoladas uma da outra, de modo que cada uma tenha a impressão de que nenhuma outra transação está sendo executada simultaneamente a ela.
  - A durabilidade garante que, quando uma transação tiver sido confirmada, as atualizações dessa transação não são perdidas, mesmo que haja uma falha no sistema.
- A execução simultânea de transações melhora o throughput de transações e a utilização do sistema, e também reduz o tempo de espera das transações.
- Quando várias transações são executadas ao mesmo tempo no banco de dados, a consistência dos dados pode não mais ser preservada. Portanto, é preciso que o sistema controle a interação entre as transações simultâneas.
  - Como uma transação é uma unidade que preserva a consistência, uma execução serial de transações garante que a consistência será preservada.
  - Um *schedule* captura as principais ações das transações que afetam a execução simultânea, como as operações `read` e `write`, enquanto abstrai detalhes internos da execução da transação.
  - Exigimos que qualquer schedule produzido pelo processamento simultâneo de um conjunto de transações terá um efeito equivalente a um schedule produzido quando essas transações são executadas serialmente em alguma ordem.
  - Considera-se que um sistema que garante essa propriedade assegura a *seriação*.
  - Existem várias noções diferentes de equivalência levando aos conceitos de *seriação de conflito* e *seriação de visão*.
- A seriação de schedules gerados pelas transações em execução simultânea pode ser garantida por meio de uma série de mecanismos chamados esquemas de *controle de concorrência*.

- Schedules precisam ser recuperáveis, para certificar que, se a transação *a* vir os efeitos da transação *b*, e depois *b* abortar, então *a* também é abortada.
- De preferência, schedules não deverão ser em cascata, de modo que o aborto de uma transação não resulte em abortos em cascata de outras transações. A não existência de cascata é assegurada permitindo-se que as transações só leiam dados confirmados.
- O componente de gerenciamento de controle simultâneo do banco de dados é responsável por lidar com os esquemas de controle de concorrência. O Capítulo 16 descreve os esquemas de controle de concorrência.
- O componente de gerenciamento de recuperação de um banco de dados é responsável por assegurar as propriedades de atomicidade e durabilidade das transações.

  O esquema de cópia de sombra é usado para garantir a atomicidade e a durabilidade nos editores de textos; porém, ele possui sobrecargas extremamente altas quando usado para sistemas de banco de dados e, além do mais, não admite a execução simultânea. O Capítulo 17 aborda esquemas melhores.
- Podemos testar determinado schedule para seriação de conflitos, construindo um *gráfico de precedência* para o schedule, e procurando a ausência de ciclos no gráfico. Porém, existem esquemas de controle de concorrência mais eficientes para garantir a seriação.

Termos de revisão

- Transação
- Propriedades ACID
  - Atomicidade
  - Consistência
  - Isolamento
  - Durabilidade
- Estado inconsistente
- Estado da transação
  - Ativa
  - Parcialmente confirmada
  - Falha
  - Abortada
  - Confirmada
  - Terminada
  - Transação
  - Reinício
  - Morta
- Escritas externas observáveis
- Esquema de cópia de sombra
- Execuções simultâneas
- Execução serial
- Schedules
- Conflito de operações
- Equivalência de conflito
- Seriação de conflito
- Equivalência de visão
- Seriação de visão
- Escritas cegas
- Facilidade de recuperação
- Schedules recuperáveis
- Rollback em cascata
- Schedules não em cascata
- Esquema de controle de concorrência
- Bloqueio
- Teste de seriação
- Gráfico de precedência
- Ordem de seriação

Exercícios práticos

15.1 Suponha que exista um sistema de banco de dados que nunca falhe. É necessário haver um gerente de recuperação para esse sistema?

15.2 Considere um sistema de arquivo como aquele no seu sistema operacional favorito.

a. Quais são os passos envolvidos na criação e exclusão de arquivos, e na gravação de dados para um arquivo?

b. Explique como as questões de atomicidade e durabilidade são relevantes para a criação e exclusão de arquivos e para a escrita de dados em arquivos.

15.3 Os implementadores de sistema de banco de dados prestaram muito mais atenção às propriedades

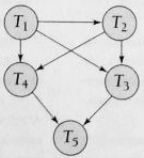


Figura 15.18 Gráfico de precedência para o Exercício prático 15.6.

ACID do que os implementadores de sistemas de arquivos. Por que isso pode ter acontecido?

15.4 Justifique a seguinte afirmação: a execução simultânea de transações é mais importante quando os dados precisam ser apanhados do disco (lento) ou quando as transações são longas, e é menos importante quando os dados estão na memória e as transações são muito curtas.

15.5 Como cada schedule serial de conflito é serial de visão, por que enfatizamos a seriação de conflito em vez da seriação de visão?

15.6 Considere o gráfico de precedência da Figura 15.16. O conflito de schedule correspondente é passível de seriação? Explique sua resposta.

15.7 O que é um schedule não em cascata? Por que se deseja que não haja schedules em cascata? Existem circunstâncias sob as quais seria desejável permitir schedules não em cascata? Explique sua resposta.

Exercícios

15.8 Liste as propriedades ACID. Explique a utilidade de cada uma.

15.9 Durante sua execução, uma transação passa por vários estados, até que finalmente confirme ou aborte. Liste todas as seqüências de estados possíveis por meio das quais uma transação pode passar. Explique por que cada transação de estado pode ocorrer.

15.10 Explique a distinção entre os termos *schedule serial* e *schedule passível de seriação*.

15.11 Considere as duas transações a seguir:

```
T1: read(A);
    read(B);
    if A = 0 then B := B + 1;
    write(B).
T2: read (B);
    read (A);
    if B = 0 then A := A + 1;
    write(A).
```

Considere que o requisito de consistência seja $A = 0 \vee B = 0$, com $A = B = 0$ sendo os valores iniciais.

a. Mostre que cada execução serial envolvendo essas duas transações preserva a consistência do banco de dados.

b. Mostre uma execução simultânea de T_1 e T_2 que produz um schedule não passível de seriação.

c. Existe uma execução simultânea de T_1 e T_2 que produza um schedule passível de seriação?

15.12 O que é um schedule recuperável? Por que a facilidade de recuperação dos schedules é desejável? Existem quaisquer circunstâncias sob as quais seria desejável permitir schedules não passíveis de recuperação? Explique sua resposta.

15.13 Por que os sistemas de banco de dados admitem a execução simultânea de transações, apesar do esforço de programação extra necessário para garantir que a execução simultânea não cause quaisquer problemas?

Notas bibliográficas

Gray e Reuter [1993] oferecem um livro–texto detalhado sobre conceitos de processamento de transação, técnicas e detalhes de implementação, incluindo o controle de concorrência e aspectos de recuperação. Bernstein e Newcomer [1997] oferecem um livro–texto sobre vários aspectos do processamento de transação.

As primeiras discussões de livro–texto sobre controle de concorrência e recuperação incluíam Bernstein *et al.* [1987] e Papadimitriou [1986]. Um antigo estudo sobre questões de implementação no controle de concorrência e recuperação é apresentado por Gray [1978].

O conceito de seriação foi formalizado por Eswaran *et al.* [1976] em conjunto com o trabalho sobre controle de concorrência para o System R. Os resultados referentes ao teste de seriação e ao problema do teste não procedural para seriação de visão são de Papadimitriou *et al.* [1977] e Papadimitriou [1979]. Os algoritmos de detecção de ciclo, além de uma introdução ao problema não procedural podem ser encontrados nos livros–texto padrão de algoritmo, como Cormen *et al.* [1990].

As referências abordando aspectos específicos de processamento de transação, como controle de concorrência e recuperação, são citadas nos Capítulos 16, 17 e 25.

CAPÍTULO 16

Controle de concorrência

Vimos, no Capítulo 15, que uma das propriedades fundamentais de uma transação é o isolamento. Contudo, quando várias transações são executadas simultaneamente no banco de dados, a propriedade de isolamento pode não ser mais preservada. Para garantir que seja, o sistema precisa controlar a interação entre as transações simultâneas; esse controle é alcançado por meio de uma série de mecanismos chamados esquemas de *controle de concorrência*.

Os esquemas de controle de concorrência que discutimos neste capítulo são todos baseados na propriedade de seriação. Ou seja, todos os esquemas apresentados aqui garantem que os schedules sejam passíveis de seriação. No Capítulo 25, discutimos os esquemas de controle de concorrência que admitem schedules não passíveis de seriação. Neste capítulo, vamos considerar o gerenciamento de transações executando simultaneamente, e ignoramos as falhas. No Capítulo 17, veremos como o sistema pode se recuperar de falhas.

Protocolos baseados em bloqueio

Um modo de garantir a seriação é exigir que os itens de dados sejam acessados de uma maneira mutuamente exclusiva; ou seja, enquanto uma transação está acessando um item de dados, nenhuma outra transação pode modificar esse item de dados. O método mais comum usado para implementar esse requisito é permitir que uma transação acesse um item de dados somente se estiver atualmente mantendo um bloqueio sobre esse item.

Bloqueios

Existem vários modos como um item de dados pode ser bloqueado. Nesta seção, restringimos nossa atenção a dois modos:

1. **Compartilhado**. Se uma transação T_i tiver obtido um **bloqueio no modo compartilhado** (indicado por S) sobre o item Q, então T_i pode ler, mas não pode escrever Q.
2. **Exclusivo**. Se uma transação T_i tiver obtido um **bloqueio no modo exclusivo** (indicado por X) sobre o item Q, então T_i pode ler e escrever Q.

É necessário que cada transação **solicite** um bloqueio em um modo apropriado sobre o item de dados Q, dependendo dos tipos de operações que realizará sobre Q. A transação faz a solicitação ao gerenciador de controle de concorrência. A transação só pode prosseguir com a operação depois que o gerenciamento de controle de concorrência **conceder** o bloqueio à transação.

Dado um conjunto de modos de bloqueio, podemos definir uma **função de compatibilidade** sobre eles da seguinte maneira. Considere que A e B representam modos de bloqueio quaisquer. Suponha que uma transação T_i solicite um bloqueio do modo A sobre o item Q, no qual a transação T_j ($T_i \neq T_j$) atualmente mantém um bloqueio do modo B. Se a transação T_i puder receber um bloqueio sobre Q imediatamente, apesar da presença do bloqueio no modo B, então dizemos que o modo A é **compatível** com o modo B. Essa função pode ser representada convenientemente por uma

	S	X
S	true	false
X	false	false

Figura 16.1 Matriz de compatibilidade de bloqueio comp.

matriz. A relação de compatibilidade entre os dois modos de bloqueio discutidos nesta seção aparece na matriz comp da Figura 16.1. Um elemento comp(A, B) da matriz tem o valor *true* (verdadeiro) se e somente se o modo A for compatível com o modo B.

Observe que o modo compartilhado é compatível com o modo compartilhado, mas não com o modo exclusivo. A qualquer momento, vários bloqueios no modo compartilhado podem ser mantidos simultaneamente (por diferentes transações) sobre um item de dados em particular. Uma solicitação de bloqueio no modo exclusivo subseqüente precisa esperar até que os bloqueios do modo compartilhado atualmente mantido sejam liberados.

Uma transação solicita um bloqueio compartilhado sobre o item de dados Q executando a instrução `lock-s`(Q). De modo semelhante, uma transação solicita um bloqueio exclusivo por meio da instrução `lock-x`(Q). Uma transação pode desbloquear um item de dados Q pela instrução `unlock`(Q).

Para acessar um item de dados, a transação T_i primeiro precisa bloquear esse item. Se o item de dados já estiver bloqueado por outra transação em um modo incompatível, o gerenciador de controle de concorrência não concederá o bloqueio até que todos os bloqueios incompatíveis mantidos por outras transações tenham sido liberados. Assim, T_i **espera** até que todos os bloqueios incompatíveis mantidos por outras transações tenham sido liberados.

A transação T_i pode desbloquear um item de dados que tinha bloqueado em algum ponto anterior. Observe que uma transação precisa manter um bloqueio sobre um item de dados desde que ele acesse esse item. Além do mais, nem sempre se deseja que uma transação desbloqueie um item de dados imediatamente após seu acesso final desse item de dados, pois a seriação pode não ser assegurada.

Como ilustração, considere novamente o sistema bancário simplificado que apresentamos no Capítulo 15. Considere que A e B sejam duas contas que são acessadas pelas transações T_1 e T_2. A transação T_1 transfere \$50 da conta B para a conta A (Figura 16.3). A transação T_2 apresenta a quantia total de dinheiro nas contas A e B – ou seja, a soma $A + B$ (Figura 16.3).

T_1: lock-$X(B)$;
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-x(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

Figura 16.2 Transação T_1.

T_2: lock-S(A);
read(A);
unlock(A);
lock-s(B);
read(B);
unlock(B);
display($A + B$).

Figura 16.3 Transação T_2.

T_1	T_2	gerenciador do controle de concorrência
lock-X(B)		
		grant-X(B,T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A,T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B,T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A,T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Figura 16.4 Schedule 1.

Suponha que os valores das contas A e B sejam $100 e $200, respectivamente. Se essas duas transações forem executadas em série, ou na ordem T_1, T_2 ou na ordem T_2, T_1, então a transação T_2 mostrará o valor $300. Se, porém, essas transações forem executadas simultaneamente, então o schedule 1, na Figura 16.4, é possível. Nesse caso, a transação T_2 mostra $250, que é incorreto. O motivo para esse engano é que a transação T_1 desbloqueou o item de dados B muito cedo e, como resultado, T_2 viu um estado inconsistente.

O schedule mostra as ações executadas pelas transações, bem como os pontos em que o gerenciador de controle de concorrência concede os bloqueios. A transação fazendo uma solicitação de bloqueio não pode executar sua próxima ação até que o gerenciador de controle de concorrência conceda o bloqueio. Logo, o bloqueio precisa ser concedido no intervalo de tempo entre a operação de solicitação de bloqueio e a ação seguinte da transação. Exatamente quando dentro desse intervalo o bloqueio será concedido não é importante; podemos seguramente considerar que o bloqueio é concedido imediatamente antes da ação seguinte da transação. Portanto, removeremos a coluna representando as ações do gerenciador de controle de concorrência de todos os schedules indicados no restante do capítulo. Deixamos para você a tarefa de deduzir quando os bloqueios são concedidos.

Suponha, agora, que o desbloqueio seja adiado para o final da transação. A transação T_3 corresponde a T_1 com des-

```
T3: lock-x(B);
    read(B);
    B := B − 50;
    write(B);
    lock-x(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
unlock(A).
```

Figura 16.5 Transação T_3.

T_4: lock-s(A);
read(A);
lock-s(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Figura 16.6 Transação T_4.

bloqueio adiado (Figura 16.5). A transação T_4 corresponde a T_2 com o desbloqueio adiado (Figura 16.6).

Você deve verificar que a sequência de leituras e escritas no schedule 1, que leva à exibição de um total incorreto de $250, não é mais possível com T_3 e T_4. Outros schedules são possíveis. T_4 não imprimirá um resultado inconsistente em qualquer um deles; veremos por que um pouco adiante.

Infelizmente, o bloqueio pode levar a uma situação indesejável. Considere o schedule parcial da Figura 16.7 para T_3 e T_4. Como T_3 está mantendo um bloqueio no modo exclusivo sobre B e T_4 está solicitando um bloqueio no modo compartilhado sobre B, T_4 está esperando que T_3 desbloqueie B. Inicialmente, como T_4 está mantendo um bloqueio no modo compartilhado sobre A e T_3 está solicitando um bloqueio no modo exclusivo sobre A, T_3 está esperando que T_4 desbloqueie A. Assim, chegamos a um estado em que nenhuma dessas transações pode prosseguir com sua execução normal. Essa situação é chamada de **impasse** (*deadlock*). Quando ocorre impasse, o sistema precisa reverter uma das duas transações. Quando uma transação tiver sido revertida, os itens de dados que estavam bloqueados por essa transação são desbloqueados. Esses itens de dados, então, ficam disponíveis para a outra transação, que pode continuar com sua execução. Retornaremos à questão do tratamento de impasse na seção "Tratamento de impasse".

Se não usarmos o bloqueio, ou se desbloquearmos itens de dados assim que for possível, depois que forem lidos ou escritos, podemos obter estados inconsistentes. Por outro lado, se não desbloquearmos um item de dados antes de solicitar um bloqueio sobre outro item de dados, os impasses poderão ocorrer. Existem maneiras de evitar o impasse em algumas situações, como veremos na seção "Protocolos baseados em grafico". Porém, em geral, os impasses são um mal necessário, associado ao bloqueio, se quisermos evitar estados inconsistentes. Os impasses são definitivamente preferíveis aos estados inconsistentes, pois podem ser tratados pelo rollback de transações, enquanto os estados inconsistentes podem levar a problemas do mundo real que não podem ser tratados pelo sistema de banco de dados.

Exigiremos que cada transação no sistema siga um conjunto de regras, chamadas **protocolo de bloqueio**, indicando quando uma transação pode bloquear e desbloquear cada um dos itens de dados. Os protocolos de bloqueio restringem o número de schedules possíveis. O conjunto de todos esses schedules é um subconjunto apropriado de todos os schedules serializáveis possíveis. Apresentaremos vários protocolos de bloqueio que permitem apenas schedules passíveis de seriação de conflito. Antes de fazer isso, precisamos de algumas definições.

Considere que $\{T_0, T_1, \ldots, T_n\}$ sejam um conjunto de transações participando de um schedule S. Dizemos que T_i **precede** T_j em S, escrito como $T_i \rightarrow T_j$, se houver um item de dados Q tal que T_i tenha mantido o modo de bloqueio A sobre Q, e T_j tenha mantido o modo de bloqueio B sobre Q mais tarde, e comp(A,B) = false. Se $T_i \rightarrow T_j$, então essa precedência implica que, em qualquer schedule serial equiva-

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figura 16.7 Schedule 2.

lente, T_i precisa aparecer antes de T_j. Observe que esse gráfico é semelhante ao gráfico de precedência que usamos na seção "Testando a seriação" do Capítulo 15 para testar a seriação de conflito. Os conflitos entre instruções correspondem à não compatibilidade de modos de bloqueio.

Dizemos que um schedule S é legal sob determinado protocolo de bloqueio se S for um schedule possível para um conjunto de transações que seguem as regras do protocolo de bloqueio. Dizemos que um protocolo de bloqueio assegura a seriação de conflito se e somente se todos os schedules legais forem passíveis de seriação de conflito; em outras palavras, para todos os schedules legais, a relação $\rightarrow$ é acíclica.

Concessão de bloqueios

Quando uma transação solicita um bloqueio sobre um item de dados em determinado modo, e nenhuma outra transação possui um bloqueio sobre o mesmo item de dados em um modo em conflito, o bloqueio pode ser concedido. Porém, deve-se ter o cuidado de evitar o cenário a seguir. Suponha que uma transação T_2 tenha um bloqueio no modo compartilhado sobre um item de dados, e outra transação T_1 solicite um bloqueio no modo exclusivo sobre o item de dados. Claramente, T_1 tem de esperar que T_2 libere o bloqueio no modo compartilhado. Nesse meio tempo, uma transação T_3 pode solicitar um bloqueio no modo compartilhado sobre o mesmo item de dados. A solicitação de bloqueio é compatível com o bloqueio concedido a T_2, de modo que T_3 pode receber o bloqueio no modo compartilhado. Nesse ponto, T_2 pode liberar o bloqueio, mas ainda T_1 precisa esperar que T_3 termine. Novamente, pode haver uma nova transação T_4 que solicite um bloqueio no modo compartilhado sobre o mesmo item de dados, recebendo o bloqueio antes que T_3 o libere. De fato, é possível que haja uma seqüência de transações em que cada uma solicite um bloqueio no modo compartilhado sobre o item de dados, e cada transação libera o bloqueio pouco depois que ele foi concedido, mas T_1 nunca recebe o bloqueio no modo exclusivo sobre o item de dados. A transação T_1 pode nunca fazer progresso, e é considerada **estagnada** (*starved*).

Podemos evitar a estagnação de transações concedendo bloqueios da seguinte maneira: quando uma transação T_i solicita um bloqueio sobre um item de dados Q em um modo específico M, o gerenciador de controle de concorrência concede o bloqueio desde que

1. Não haja outra transação mantendo um bloqueio sobre Q em um modo que entra em conflito com M.
2. Não existe outra transação que esteja esperando por um bloqueio sobre Q e que fez sua solicitação de bloqueio antes de T_i.

Assim, uma solicitação de bloqueio nunca será bloqueada por uma solicitação de bloqueio que for feita mais tarde.

O protocolo de bloqueio em duas fases

Um protocolo que assegura a seriação é o protocolo de bloqueio em duas fases. Esse protocolo requer que cada transação emita solicitações de bloqueio e desbloqueio em duas fases:

1. **Fase de crescimento.** Uma transação pode obter bloqueios, mas não pode liberar qualquer bloqueio.
2. **Fase de encolhimento.** Uma transação pode liberar bloqueios, mas não pode obter novos bloqueios.

Inicialmente, uma transação está na fase de crescimento. A transação adquire bloqueios conforme a necessidade. Quando a transação libera um bloqueio, ele entra na fase de encolhimento e não pode emitir mais solicitações de bloqueio.

Por exemplo, as transações T_3 e T_4 são de duas fases. Por outro lado, as transações T_1 e T_2 não são de duas fases. Observe que as instruções de desbloqueio não precisam aparecer no final da transação. Por exemplo, no caso da transação T_3, poderíamos mover a instrução `unlock`(B) para logo depois da instrução `lock-x`(A), e ainda reter a propriedade de bloqueio em duas fases.

Podemos mostrar que o protocolo de bloqueio em duas fases assegura a seriação de conflito. Considere qualquer transação. O ponto no schedule em que a transação obteve seu bloqueio final (o final de sua fase de crescimento) é denominado **ponto de bloqueio** da transação. Agora, as transações podem ser ordenadas de acordo com seus pontos de bloqueio – essa ordenação, na verdade, é uma ordenação de seriação para as transações. Deixamos a prova como um exercício para você fazer (ver Exercício prático 16.1).

O bloqueio em duas fases *não* garante a liberdade do impasse. Observe que as transações T_3 e T_4 são de duas fases, mas, no schedule 2 (Figura 16.7), elas estão em impasse.

Lembre-se, pela seção "Schedules não em cascata" do Capítulo 15, que, além de serem passíveis de seriação, os schedules não devem ser em cascata. O rollback em cascata pode ocorrer sob o bloqueio em duas fases. Como ilustração, considere o schedule parcial da Figura 16.8. Cada transação observa o protocolo de bloqueio em duas fases, mas a falha de T_5 após a etapa `read`(A) de T_7 leva a um rollback em cascata de T_6 e T_7.

Os rollbacks em cascata podem ser evitados por uma modificação do bloqueio em duas fases chamado **protocolo de bloqueio estrito em duas fases**. Esse protocolo requer não apenas que o bloqueio seja em duas fases, mas também que todos os bloqueios no modo exclusivo realizados por uma transação sejam mantidos até que essa transação seja

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figura 16.8 Schedule parcial sob o bloqueio em duas fases.

confirmada. Esse requisito garante que quaisquer dados escritos por uma transação não confirmada sejam bloqueados no modo exclusivo até que a transação seja confirmada, evitando que qualquer outra transação leia os dados.

Outra variante do bloqueio em duas fases é o **protocolo de bloqueio rigoroso em duas fases**, que exige que todos os bloqueios sejam mantidos até que a transação seja confirmada. Podemos facilmente verificar que, com o bloqueio rigoroso em duas fases, as transações podem ser seriadas na ordem em que são confirmadas. A maioria dos sistemas de banco de dados implementa o bloqueio estrito ou rigoroso em duas fases.

Considere as duas transações a seguir, para as quais mostramos apenas algumas das operações read e write significativas:

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display(a_1 + a_2).

Se empregarmos o protocolo de bloqueio em duas fases, então T8 terá de bloquear a1 no modo exclusivo. Portanto, qualquer execução simultânea das duas transações leva a uma execução serial. Porém, observe que T8 precisa de um bloqueio exclusivo sobre a_1 somente no final de sua execução, quando escreve a_1. Assim, se T_8 pudesse inicialmente bloquear a_1 no modo compartilhado, e então mais tarde pudesse alterar o bloqueio para o modo exclusivo, poderíamos obter mais concorrência, pois T_8 e T_9 poderiam acessar a_1 e a_2 simultaneamente.

Essa observação nos leva a um refinamento do protocolo de bloqueio básico em duas fases, em que as **conversões de bloqueio** são permitidas. Vamos oferecer um mecanismo para elevar um bloqueio compartilhado para um bloqueio exclusivo e reduzir um bloqueio exclusivo para um bloqueio compartilhado. Indicamos a conversão do modo compartilhado para o exclusivo por **upgrade**, e do exclusivo para o compartilhado por **downgrade**. A conversão de bloqueio não pode ser permitida arbitrariamente. Em vez disso, o processo de upgrade pode ocorrer apenas na fase de crescimento, enquanto o downgrade pode ocorrer apenas na fase de encolhimento.

Retornando ao nosso exemplo, as transações T_8 e T_9 podem ser executadas simultaneamente sob o protocolo de

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

bloqueio de duas fases refinado, como mostra o schedule incompleto da Figura 16.9, em que só aparece uma das instruções de bloqueio.

Observe que uma transação tentando fazer o upgrade de um bloqueio sobre um item Q pode ser forçada a esperar. Essa espera forçada ocorre se Q estiver atualmente bloqueado por *outra* transação no modo compartilhado.

Assim como o protocolo de bloqueio básico em duas fases, o bloqueio em duas fases com conversão de bloqueio gera apenas schedules seriais de conflito, e as transações podem ser colocadas em série por seus pontos de bloqueio. Além disso, se bloqueios exclusivos forem mantidos até o final da transação, os schedules não são de cascata.

Para um conjunto de transações, pode haver schedules seriais de conflito, que não podem ser obtidos por meio do protocolo de bloqueio em duas fases. Porém, para obter schedules seriais de conflito por protocolos de bloqueio não em duas fases, precisamos ter informações adicionais sobre as transações ou impor alguma estrutura ou ordenação sobre o conjunto de itens de dados no banco de dados. Na ausência de tal informação, o bloqueio em duas fases é necessário para seriação de conflito – se T_i é uma transação não em duas fases, sempre é possível encontrar outra transação T_j que seja de duas fases, para que haja um schedule possível para T_i e T_j, que não seja passível de seriação de conflito.

O bloqueio estrito em duas fases e o bloqueio rigoroso em duas fases (com conversões de bloqueio) são usados extensivamente nos sistemas de banco de dados comerciais.

Um esquema simples, mas bastante utilizado, gera automaticamente as instruções apropriadas de bloqueio e desbloqueio para uma transação, com base nas solicitações de leitura e escrita da transação:

- Quando uma transação T_i emite uma operação `read`(Q), o sistema emite uma instrução `lock-s`(Q) seguida pela instrução `read`(Q).
- Quando T_i emite uma operação `write`(Q), o sistema verifica se T_i já mantém um bloqueio compartilhado sobre Q. Se mantiver, então o sistema emite uma instrução `upgrade`(Q), seguida pela instrução `write`(Q). Caso contrário, o sistema emite uma instrução `lock-x`(Q), seguida pela instrução `write`(Q).
- Todos os bloqueios obtidos por uma transação são desbloqueados depois que a transação é confirmada ou abortada.

*Implementação do bloqueio***

Um **gerenciador de bloqueio** pode ser implementado como um processo que recebe mensagens das transações e envia mensagens em resposta. O processo do gerenciador de bloqueio responde a mensagens de solicitação de bloqueio com mensagens de concessão de bloqueio ou com mensagens solicitando o rollback da transação (no caso de impasses). As mensagens de desbloqueio exigem apenas uma confirmação em resposta, mas podem resultar em uma mensagem de concessão para outra transação aguardando.

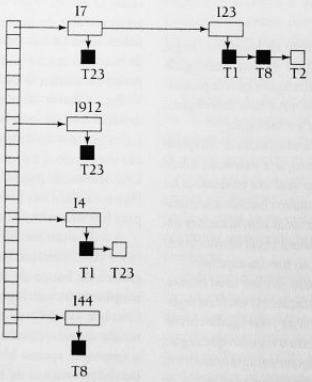


Figura 16.10 Tabela de bloqueio.

O gerenciador de bloqueio utiliza essa estrutura de dados: para cada item de dados que está atualmente bloqueado, ele mantém uma lista interligada de registros, um para cada solicitação, na ordem em que as solicitações chegaram. Ele usa uma tabela de hash, indexada sobre o nome de um item de dados, para encontrar a lista interligada (se houver) para um item de dados; essa tabela é chamada **tabela de bloqueio**. Cada registro da lista interligada para um item de dados anota qual transação fez a solicitação e qual modo de bloqueio foi solicitado. O registro também anota se a solicitação atualmente foi concedida.

A Figura 16.10 mostra um exemplo de uma tabela de bloqueio. A tabela contém bloqueios para cinco diferentes itens de dados, I4, I7, I23, I44 e I912. A tabela de bloqueio usa o encadeamento de estouro, de modo que existe uma lista interligada dos itens de dados para cada entrada na tabela de bloqueio. Há também uma lista de transações que receberam bloqueios, ou que estão esperando por bloqueios, para cada um dos itens de dados. Os bloqueios concedidos são os retângulos preenchidos (pretos), enquanto as solicitações aguardando são os retângulos vazios. Omitimos o modo de bloqueio para manter a figura simples. Podemos ver, por exemplo, que T23 recebeu bloqueios sobre I912 e I7, e está esperando por um bloqueio sobre I4.

Embora a figura não mostre isso, a tabela de bloqueio também deve manter um índice sobre identificadores de transação, de modo que é possível determinar de forma eficiente o conjunto de bloqueios mantidos por determinada transação.

O gerenciador de bloqueio processa solicitações desta maneira:

- Quando a mensagem de solicitação de bloqueio chega, ele acrescenta um registro ao final da lista interligada para o item de dados, se a lista interligada estiver presente. Caso contrário, ela cria uma nova lista interligada, contendo apenas o registro para a solicitação.

  Ele sempre concede a primeira solicitação de bloqueio sobre um item de dados. Entretanto, se a transação solicitar um bloqueio sobre um item no qual um bloqueio já foi concedido, o gerenciador de bloqueio concede a solicitação apenas se for compatível com todas as solicitações anteriores, e todas as solicitações anteriores já foram concedidas. Caso contrário, a solicitação precisa esperar.
- Quando o gerenciador de bloqueio receber uma mensagem de desbloqueio de uma transação, ele exclui o registro para esse item de dados na lista interligada correspondente a essa transação. Ele testa o registro que segue, se houver, conforme descrevemos no parágrafo anterior, para ver se essa solicitação agora pode ser concedida. Se puder, o gerenciador de bloqueio concede essa solicitação e processa o registro depois dele, se houver, de modo semelhante, e assim por diante.
- Se uma transação abortar, o gerenciador de bloqueio exclui qualquer solicitação esperando feita pela transação. Quando o sistema de banco de dados tiver tomado ações apropriadas para desfazer a transação (ver seção "Recuperação e atomicidade" do Capítulo 17), ele libera todos os bloqueios mantidos pela transação abortada.

Esse algoritmo garante liberdade de estagnação para as solicitações de bloqueio, pois uma solicitação nunca poderá ser concedida enquanto uma solicitação recebida anteriormente estiver aguardando concessão. Estudaremos como detectar e tratar com impasses depois, na seção "Detecção e recuperação de impasse". A seção "Estrutura do processo servidor de transações" do Capítulo 20 descreve uma implementação alternativa – que utiliza memória compartilhada no lugar da passagem de mensagem para solicitação/concessão de bloqueio.

Protocolos baseados em gráfico

Conforme observado na seção "O protocolo de bloqueio em duas fases", o protocolo de bloqueio em duas fases é necessário e suficiente para garantir a seriação na ausência de informações com relação à maneira em que os itens de dados são acessados. Todavia, se quisermos desenvolver protocolos que não são de duas fases, precisamos de informações adicionais sobre como cada transação acessará o banco de dados. Existem vários modelos que nos dão as informações adicionais, cada um diferente na quantidade de informações fornecidas. O modelo mais simples exige conhecimento anterior sobre a ordem em que os itens do banco de dados serão acessados. Com tal informação, é possível construir protocolos de bloqueio que não são de duas fases, mas que, apesar disso, garantem a seriação de conflito.

Para adquirir tal prioridade anterior, impomos uma ordenação parcial $\rightarrow$ sobre o conjunto $\mathbf{D} = \{d_1, d_2, \ldots, d_h\}$ de todos os itens de dados. Se $d_i \rightarrow d_j$, então qualquer transação acessando d_i e d_j precisam acessar d_i antes de acessar d_j. Essa ordenação parcial pode ser o resultado da organização lógica ou física dos dados, ou pode ser imposta unicamente para fins de controle de concorrência.

A ordenação parcial implica que o conjunto **D** pode agora ser visto como um gráfico acíclico direcionado, chamado **gráfico de banco de dados**. Nesta seção, por motivos de simplicidade, restringiremos nossa atenção apenas aos gráficos que são árvores enraizadas. Apresentaremos um protocolo simples, chamado *protocolo de árvore*, que é restrito a empregar apenas bloqueios *exclusivos*. As referências a outros protocolos de bloqueio mais complexos, baseados em gráfico, estão nas notas bibliográficas.

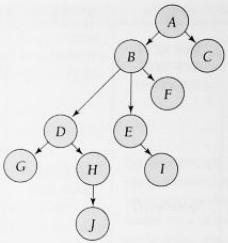


Figura 16.11 Gráfico de banco de dados estruturado em árvore.

No **protocolo de árvore**, a única instrução de bloqueio permitida é `lock-x`. Cada transação T_i pode bloquear um item de dados no máximo uma vez, e precisa observar as seguintes regras:

1. O primeiro bloqueio por T_i pode ser sobre qualquer item de dados.
2. Subsequentemente, um item de dados Q pode ser bloqueado por T_i somente se o pai de Q estiver atualmente bloqueado por T_i.
3. Os itens de dados podem ser desbloqueados a qualquer momento.
4. Um item de dados que foi bloqueado e desbloqueado por T_i não pode mais tarde ser bloqueado novamente por T_i.

Todos os schedules que são legais sob o protocolo de árvore são seriais de conflito.

Para ilustrar esse protocolo, considere o gráfico de banco de dados da Figura 16.11. As quatro transações a seguir seguem o protocolo de árvore sobre esse gráfico. Mostramos apenas as instruções de bloqueio e desbloqueio:

T_{10}: `lock-x`(B); `lock-x`(E), `lock-x`(D); `unlock-x`(B), `unlock`(E); `lock-x`(G); `unlock`(D); `unlock`(G).
T_{11}: `lock-x`(D); `lock-x`(H); `unlock`(D); `unlock`(H).
T_{12}: `lock-x`(B); `lock-x`(E); `unlock`(E); `unlock`(B).
T_{13}: `lock-x`(D); `lock-x`(H); `unlock`(D); `unlock`(H).

Um schedule possível, em que essas quatro transações participaram, aparece na Figura 16.12. Observe que, durante sua execução, a transação T_{10} mantém bloqueios sobre duas subárvores *disjuntas*.

Observe que o schedule da Figura 16.12 é passível de seriação de conflito. Pode-se mostrar não apenas que o protocolo de árvore garante a seriação de conflito, mas também que esse protocolo assegura liberdade de impasse.

O protocolo de árvore na Figura 16.12 não garante a facilidade de recuperação e inexistência de cascata. Para garanti-las, o protocolo pode ser modificado de modo a não permitir liberação de bloqueios exclusivos até o final da transação. Manter bloqueios exclusivos até o final da transação reduz a concorrência. Aqui está uma alternativa que melhora a concorrência, mas garante apenas a facilidade de recuperação: para cada item de dados com uma escrita não confirmada, registramos qual transação realizou a última escrita no item de dados. Sempre que uma transação T_i realiza uma leitura de um item de dados não confirmado, registramos uma **dependência de commit** de T_i sobre a transação que realizou a última escrita no item de dados. A transação T_i, então, não tem permissão para confirmar antes do commit de todas as transações em que possui uma dependência de commit. Se qualquer uma dessas transações for abortada, T_i também terá de ser abortado.

O protocolo de bloqueio em árvore tem uma vantagem em relação ao protocolo de bloqueio em duas fases porque, diferente do bloqueio em duas fases, ele é livre de impasse, de modo que nenhum rollback é exigido. O protocolo de bloqueio em árvore tem outra vantagem em relação ao protocolo de bloqueio em duas fases porque o desbloqueio pode ocorrer mais cedo. O desbloqueio mais cedo pode ocasionar menores tempos de espera e um aumento na concorrência.

Entretanto, o protocolo tem a desvantagem de que, em alguns casos, uma transação pode ter de bloquear itens de dados que ele não acessa. Por exemplo, uma transação que precisa acessar itens de dados A e J no gráfico de banco de dados da Figura 16.11 precisa bloquear não apenas A e J, mas também os itens de dados B, D e H. Esse bloqueio adicional resulta em maior sobrecarga de bloqueio, a possibili-

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

Figura 16.12 Schedule serial sob o protocolo de árvore.

dade de tempo de espera adicional e uma diminuição em potencial na concorrência. Além do mais, sem conhecimento anterior de quais itens de dados precisarão ser bloqueados, as transações terão de bloquear a raiz da árvore, e isso pode reduzir muito a concorrência.

Para um conjunto de transações, pode haver schedules passíveis de seriação de conflito, que não podem ser obtidos por meio do protocolo de árvore. Na realidade, existem schedules possíveis sob o protocolo de bloqueio em duas fases que não são possíveis sob o protocolo de árvore, e vice-versa. Alguns exemplos desses schedules são explorados nos exercícios.

Protocolos baseados em timestamp

Os protocolos de bloqueio que descrevemos até aqui determinam a ordem entre cada par de transações em conflito no momento da execução do primeiro bloqueio que os dois membros do par solicitam envolvendo modos incompatíveis. Outro método de determinar a ordem de seriação é selecionar uma ordenação entre as transações com antecedência. O método mais comum para fazer isso é usar um esquema de *ordenação por timestamp* .

Timestamp

Com cada transação Ti no sistema, associamos um timestamp fixo exclusivo, indicado por TS(T_i). Esse timestamp é atribuído pelo sistema de banco de dados antes que a transação T_i inicie sua execução. Se uma transação T_i tiver recebido o timestamp TS(T_i), e uma nova transação T_j entrar no sistema, então TS(T_i) < TS(T_j). Existem dois métodos simples para implementar esse esquema:

1. Use o valor do **clock do sistema** como timestamp; ou seja, o timestamp de uma transação é igual ao valor do clock quando a transação entrar no sistema.
2. Use um **contador lógico** que é incrementado após um novo timestamp ter sido atribuído; ou seja, o timestamp de uma transação é igual ao valor do contador quando a transação entrar no sistema.

Os timestamps das transações determinam a ordem de seriação. Assim, se TS(T_i) < TS(T_j), então o sistema precisa garantir que o schedule produzido é equivalente a um schedule serial em que a transação T_i aparece antes da transação T_j.

Para implementar esse esquema, associamos a cada item de dados Q dois valores de timestamp:

- **W-timestamp**(Q) indica o maior timestamp de qualquer transação que executou write(Q) com sucesso.
- **R-timestamp**(Q) indica o maior timestamp de qualquer transação que executou read(Q) com sucesso.

Esses timestamps são atualizados sempre que uma nova instrução read(Q) ou write(Q) é executada.

O protocolo de ordenação por timestamp

O protocolo de ordenação de timestamp garante que quaisquer operações read e write em conflito sejam executadas em ordem de timestamp. Esse protocolo opera da seguinte forma:

1. Suponha que a transação T_i emita read(Q).
   a. Se $TS(T_i) < W\text{-timestamp}(Q)$, então T_i precisa ler um valor de Q que já foi modificado por outro. Logo, a operação read é rejeitada, e T_i é revertida.
   b. Se $TS(T_i) \geq W\text{-timestamp}(Q)$, então a operação read é executada, e R-timestamp(Q) é definido como maior dentre R-timestamp(Q) e $TS(T_i)$.
2. Suponha que a transação T_i emita write(Q).
   a. Se $TS(T_i) < R\text{-timestamp}(Q)$, então o valor de Q que T_i está produzindo foi necessário anteriormente, e o sistema considerou que esse valor nunca seria produzido. Logo, o sistema rejeita a operação write e reverte T_i.
   b. Se $TS(T_i) < W\text{-timestamp}(Q)$, então Ti está tentando escrever um valor absoluto de Q. Logo, o sistema rejeita essa operação write e reverte T_i.
   c. Caso contrário, o sistema executa a operação write e define W-timestamp(Q) como $TS(T_i)$.

Se uma transação T_i for revertida pelo esquema de controle de concorrência como resultado da emissão de uma operação read ou write, o sistema lhe atribui um novo timestamp e o reinicia.

Para ilustrar esse protocolo, consideramos as transações T_{14} e T_{15}. A transação T_{14} apresenta o conteúdo das contas A e B:

T_{14}: read(B);
read(A);
display($A + B$).

A transação T15 transfere $50 da conta B para a conta A, e depois apresenta o conteúdo de ambas:

T_{15}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

Na apresentação de schedules sob o protocolo de timestamp, vamos considerar que uma transação recebeu um timestamp imediatamente antes de sua primeira instrução. Assim, no schedule 3 da Figura 16.13, $TS(T_{14}) < TS(T_{15})$, e o schedule é possível sob o protocolo de timestamp.

Observamos que a execução anterior também pode ser produzida pelo protocolo de bloqueio em duas fases. Porém, existem schedules que são possíveis sob o protocolo de bloqueio em duas fases, mas não são possíveis sob o protocolo de timestamp, e vice-versa (ver Exercício 16.25).

O protocolo de ordenação por timestamp garante a seriação de conflito. Isso porque as operações conflitantes são processadas em ordem de timestamp.

O protocolo garante liberdade de impasse, pois nenhuma transação sequer espera. Porém, existe uma possibilidade de estagnação de transações longas, se uma seqüência de transações curtas em conflito causar o reinício repetido da transação longa. Se uma transação for descoberta sendo reiniciada repetidamente, as transações em conflito precisam ser temporariamente bloqueadas para permitir que a transação termine.

O protocolo pode gerar schedules que não são recuperáveis. Porém, ele pode ser estendido para tornar os schedules recuperáveis, em uma dentre várias maneiras:

- A facilidade de recuperação e inexistência de cascata podem ser asseguradas realizando todas as escritas juntas no final da transação. As escritas precisam ser atômicas no seguinte sentido: enquanto as escritas estão em andamento, nenhuma transação tem permissão para acessar qualquer um dos itens de dados que foram escritos.

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Figura 16.13 Schedule 3.

- A facilidade de recuperação e inexistência de cascata também podem ser asseguras usando uma forma limitada de bloqueio, pela qual as leituras de itens não confirmados são adiadas até que a transação que atualizou o item seja confirmada (ver Exercício 16.26).
- A facilidade de recuperação isoladamente pode ser assegurada acompanhando as escritas não confirmadas, e permitindo que uma transação T_i seja confirmada apenas depois do commit de qualquer transação que escreveu um valor que T_i leu. As dependências de commit, esboçadas na seção "Protocolos baseados em gráfico", podem ser usadas para esse propósito.

Regra do write de Thomas

Agora, apresentamos uma modificação do protocolo de ordenação de timestamp que permite maior concorrência em potencial do que o protocolo da seção "O protocolo de ordenação por timestamp". Vamos considerar o schedule 4 da Figura 16.14 e aplicar o protocolo de ordenação de timestamp. Como T_{16} começa antes de T_{17}, vamos supor que $TS(T_{16}) < TS(T_{17})$. A operação `read`(Q) de T_{16} tem sucesso, assim como a operação `write`(Q) de T_{17}. Quando T_{16} tenta realizar sua operação `write`(Q), descobrimos que $TS(T_{16}) <$ W-timestamp(Q), pois W-timestamp(Q) = $TS(T_{17})$. Assim, o `write`(Q) por T_{16} é rejeitado e a transação T_{16} precisa ser revertida.

Embora o rollback de T_{16} seja exigido pelo protocolo de ordenação de timestamp, ele é desnecessário. Como T_{17} já escreveu Q, o valor que T_{16} está tentando escrever é aquele que nunca precisará ser lido. Qualquer transação T_i com $TS(T_i) < TS(T_{17})$ que tente realizar um `read`(Q) será revertida, pois $TS(T_i) <$ W-timestamp(Q). Qualquer transação T_j com $TS(T_j) > TS(T_{17})$ precisa ler o valor de Q escrito por T_{17}, em vez do valor escrito por T_{16}.

Essa observação leva a uma versão modificada do protocolo de ordenação de timestamp, em que operações write obsoletas podem ser ignoradas sob certas circunstâncias. As regras de protocolo para operações read permanecem inalteradas. As regras de protocolo para operações write, porém, são ligeiramente diferentes do protocolo de ordenação de timestamp da seção "O protocolo de ordenação por timestamp".

A modificação no protocolo de ordenação por timestamp, chamada **regra do write de Thomas**, é esta: suponha que a transação T_i emita `write`(Q).

1. Se $TS(T_i) <$ R-timestamp(Q), então o valor de Q que T_i está produzindo foi necessário anteriormente, e assumiu-se que o valor nunca seria produzido. Logo, o sistema rejeita a operação `write` e reverte T_i.
2. Se $TS(T_i) <$ W-timestamp(Q), então T_i está tentando escrever um valor obsoleto de Q. Logo, essa operação `write` pode ser ignorada.
3. Caso contrário, o sistema executa a operação `write` e define W-timestamp(Q) como $TS(T_i)$.

A diferença entre essas regras e aquelas da seção "O protocolo de ordenação por timestamp" está na segunda regra. O protocolo de ordenação de timestamp requer que T_i seja revertido se T_i emitir `write`(Q) e $TS(T_i) <$ W-timestamp(Q). Contudo, aqui, naqueles casos em que $TS(T_i) \geq$ R-timestamp(Q), ignoramos o `write` obsoleto.

A regra do write de Thomas utiliza a seriação de visão, com efeito, excluindo operações write obsoletas das transações que as emitem. Essa modificação de transações possibilita gerar schedules passíveis de seriação que não seriam possíveis sob os outros protocolos apresentados neste capítulo. Por exemplo, o schedule 4 da Figura 16.14 não é passível de seriação de conflito e, assim, não é possível sob o protocolo de bloqueio em duas fases, o protocolo de árvore e o protocolo de ordenação por timestamp. Sob a regra do write de Thomas, a operação `write`(Q) de T_{16} seria ignorada. O resultado é um schedule que é equivalente em visão ao schedule serial $<T_{16}, T_{17}>$.

Protocolos baseados em validação

Nos casos em que a maioria das transações são transações somente leitura, a taxa de conflitos entre as transações pode ser baixa. Mesmo assim, muitas dessas transações, se executadas sem a supervisão de um esquema de controle de concorrência, deixariam o sistema em um estado consistente. Um esquema de controle de concorrência impõe sobrecarga de execução de código e possível atraso das transações. Pode ser melhor usar um esquema alternativo que impõe menos overhead. Uma dificuldade na redução do overhead é que não sabemos com antecedência quais transações estarão envolvidas em um conflito. Para obter esse conhecimento, precisamos de um esquema para **monitorar** o sistema.

Consideramos que cada transação T_i é executada em duas ou três fases diferentes em seu tempo de vida, depen-

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

Figura 16.14 Schedule 4.

dendo se ela é somente de leitura ou uma transação de atualização. As fases são, em ordem,

1. **Fase de leitura**. Durante essa fase, o sistema executa a transação T_i. Ele lê os valores dos diversos itens de dados e os armazena em variáveis locais a T_i. Ele realiza todas as operações write em variáveis locais temporárias, sem atualizações do banco de dados real.
2. **Fase de validação**. A transação T_i realiza um teste de validação para determinar se pode copiar para o banco de dados as variáveis locais temporárias que mantêm os resultados das operações write sem causar violação da seriação.
3. **Fase de escrita**. Se a transação T_i tiver sucesso na validação (etapa 2), então o sistema aplica as atualizações reais ao banco de dados. Caso contrário, o sistema reverte T_i.

Cada transação precisa passar por três fases na ordem indicada. Porém, todas as três fases das transações executando simultaneamente podem ser intercaladas.

Para realizar o teste de validação, precisamos saber quando as várias fases das transações T_i ocorreram. Portanto, associaremos três timestamps diferentes com a transação T_i:

1. **Start**(T_i), o momento em que T_i iniciou sua execução.
2. **Validation**(T_i), o momento em que T_i terminou sua fase de leitura e iniciou sua fase de validação.
3. **Finish**(T_i), o momento em que T_i terminou sua fase de escrita.

Determinamos a ordem de seriação pela técnica de ordenação de timestamp, usando o valor do timestamp Validation(T_i). Assim, o valor TS(T_i) = Validation(T_i) e, se TS(T_j) < TS(T_k), então qualquer schedule produzido terá de ser equivalente a um schedule serial em que a transação T_j aparece antes da transação T_k. O motivo de termos escolhido Validation(T_i), em vez de Start(T_i), como timestamp da transação T_i é que podemos esperar tempos de resposta mais rápidos, visto que as taxas de conflito entre as transações são realmente baixas.

O **teste de validação** para a transação T_j exige que, para todas as transações T_i com TS(T_i) < TS(T_j), uma das duas condições a seguir precisam ser mantidas:

1. Finish(T_i) < Start(T_j). Como T_i completa sua execução antes que T_j inicie, a ordem de seriação é realmente mantida.
2. O conjunto de itens de dados escritos por T_i não tem interseção com o conjunto de itens de dados lidos por T_j, e T_i completa sua fase de escrita antes que T_j inicie sua fase de validação (Start(T_j) < Finish(T_i) < Validation(T_j)). Essa condição garante que as escritas de T_i e T_j não sejam sobrepostas. Como as escritas de T_i não afetam a leitura de T_j, e como T_j não pode afetar a leitura de T_i, a ordem de seriação é realmente mantida.

Como ilustração, considere novamente as transações T_{14} e T_{15}. Suponha que TS(T_{14}) < TS(T_{15}). Então, a fase de validação tem sucesso no schedule 5 da Figura 16.15. Observe que as escritas nas variáveis reais são realizadas somente após a fase de validação de T_{15}. Assim, T_{14} lê os valores antigos de B e A, e esse schedule é passível de seriação.

O esquema de validação protege automaticamente contra rollbacks em cascata, pois as escritas reais ocorrem somente depois que a transação emitindo a escrita tenha sido confirmada. Porém, existe uma possibilidade de estagnação de transações longas, devido a uma seqüência de transações curtas em conflito, o que causa reinícios repetidos da transação longa. Para evitar a estagnação, as transações em conflito precisam ser temporariamente bloqueadas, para permitir que a transação longa termine.

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
⟨*validar*⟩	
display($A + B$)	
	⟨*validar*⟩
	write(B)
	write(A)

Figura 16.15 Schedule 5, um schedule produzido pelo uso da validação.

Esse esquema de validação é denominado esquema de **controle de concorrência otimista**, pois as transações são executadas de forma otimista, supondo que serão capazes de concluir a execução e validar no final. Ao contrário, o bloqueio e a ordenação por timestamp são pessimistas, pois forçam uma espera ou um rollback sempre que um conflito é detectado, embora exista uma chance de que o schedule possa ser passível de seriação de conflito.

Granularidade múltipla

Nos esquemas de controle de concorrência descritos até aqui, usamos cada item de dados individual como unidade em que o sincronismo é realizado.

Contudo, existem circunstâncias em que seria vantajoso agrupar vários itens de dados e tratá-los como uma unidade de sincronismo individual. Por exemplo, se uma transação T_i precisar acessar o banco de dados inteiro, e um protocolo de bloqueio for utilizado, então T_i precisa bloquear cada item no banco de dados. Claramente, a execução desses bloqueios é demorada. Seria melhor se T_i pudesse emitir uma única solicitação de bloqueio para bloquear o banco de dados inteiro. Por outro lado, se a transação T_j precisar acessar apenas alguns itens de dados, ela não teria de bloquear o banco de dados inteiro, pois dessa forma a concorrência seria perdida.

O que se faz necessário é um mecanismo para permitir que o sistema defina vários níveis de **granularidade**. Podemos criar um permitindo que os itens de dados sejam de vários tamanhos e definindo uma hierarquia de granularidades de dados, em que as granularidades pequenas são aninhadas dentro das maiores. Essa hierarquia pode ser representada graficamente como uma árvore. Observe que a árvore que descrevemos aqui é significativamente diferente daquela usada pelo protocolo de árvore (seção "Protocolos baseados em gráfico"). Um nó não de folha da árvore de granularidade múltipla representa os dados associados aos seus descendentes. No protocolo de árvore, cada nó é um item de dados independente.

Como ilustração, considere a árvore da Figura 16.16, que consiste em quatro níveis de nós. O nível mais alto representa o banco de dados inteiro. Abaixo dele estão os nós do tipo *área*; o banco de dados consiste exatamente nessas áreas. Cada área, por sua vez, tem dois nós do tipo *arquivo* como seus filhos. Cada área contém exatamente aqueles arquivos que são seus nós filhos. Nenhum arquivo está em mais de uma área. Finalmente, cada arquivo tem nós do tipo *registro*. Como antes, o arquivo consiste exatamente naqueles registros que são seus nós filhos, e nenhum registro pode estar presente em mais de um arquivo.

Cada nó na árvore pode ser bloqueado individualmente. Como fizemos no protocolo de bloqueio em duas fases, usaremos os modos de bloqueio **compartilhado** e **exclusivo**. Quando uma transação bloqueia um nó, seja no modo compartilhado ou exclusivo, a transação também implicitamente terá bloqueado todos os descendentes desse nó no mesmo modo de bloqueio. Por exemplo, se a transação T_i receber um **bloqueio explícito** sobre o arquivo F_c da Figura 16.16, no modo exclusivo, então terá um **bloqueio implícito** no modo exclusivo sobre todos os registros pertencentes a esse arquivo. Ela não precisa bloquear os registros individuais de F_c explicitamente.

Suponha que a transação T_j queira bloquear o registro r_{b_2} do arquivo F_b. Como T_i bloqueou F_b explicitamente, segue-se que r_{b_2} também é bloqueado (implicitamente). Todavia, quando T_j emite uma solicitação de bloqueio para r_{b_2}, r_{b_2} não é explicitamente bloqueada! Como o sistema determina se T_j pode bloquear r_{b_2}? T_j precisa atravessar a árvore pela raiz até o registro r_{b_2}. Se qualquer nó nesse caminho for bloqueado em um modo incompatível, então T_j precisa ser adiada.

Suponha agora que a transação T_k queira bloquear o banco de dados inteiro. Para isso, ela simplesmente precisa bloquear a raiz da hierarquia. Observe, porém, que T_k não

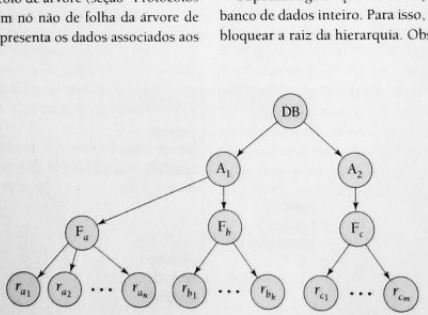


Figura 16.16 Hierarquia de granularidade.

deverá ter sucesso no bloqueio do nó raiz, pois T_i atualmente está mantendo um bloqueio sobre parte da árvore (especificamente, sobre o arquivo F_b). Porém, como o sistema determina se o nó raiz pode ser bloqueado? Uma possibilidade é que ele pesquise a árvore inteira. No entanto, essa solução vai contra o propósito inteiro do esquema de bloqueio por granularidade múltipla. Um modo mais eficiente de obter esse conhecimento é introduzir uma nova classe de modos de bloqueio, chamada **modos de bloqueio de intenção**. Se um nó for bloqueado em um modo de intenção, o bloqueio explícito está sendo feito em um nível mais baixo da árvore (ou seja, em uma granularidade mais fina). Os bloqueios de intenção são colocados em todos os ancestrais de um nó antes que esse nó seja bloqueado explicitamente. Assim, uma transação não precisa pesquisar a árvore inteira para determinar se ela pode bloquear um nó com sucesso. Uma transação que deseja bloquear um nó – digamos, Q – precisa atravessar um caminho na árvore da raiz até Q. Enquanto atravessa a árvore, a transação bloqueia os diversos nós em um modo de intenção.

Há um modo de intenção associado ao modo compartilhado, e outro com modo exclusivo. Se um nó for bloqueado no **modo compartilhado de intenção** (IS – Intention-Shared), o bloqueio explícito está feito em um nível mais baixo da árvore, mas apenas com bloqueios no modo compartilhado. De modo semelhante, se um nó for bloqueado no **modo exclusivo de intenção** (IX – Intention-eXclusive), então o bloqueio explícito está sendo feito em um nível mais baixo, com bloqueios no modo exclusivo ou compartilhado. Finalmente, se um nó estiver bloqueado no **modo compartilhado e exclusivo de intenção** (SIX – Shared and Intention-eXclusive), a subárvore que tem esse nó como raiz é bloqueada explicitamente no modo compartilhado, e esse bloqueio explícito está sendo feito em um nível mais baixo com bloqueios no modo exclusivo. A função de compatibilidade para esses modos de bloqueio está na Figura 16.17.

O **protocolo de bloqueio de granularidade múltipla**, que assegura a seriação, é este: cada transação T_i pode bloquear um nó Q seguindo estas regras:

1. Precisa observar a função de compatibilidade de bloqueio da Figura 16.17.
2. Precisa bloquear a raiz da árvore primeiro, e pode bloqueá-la em qualquer modo.
3. Pode bloquear um nó Q no modo S (compartilhado) ou IS somente se atualmente tiver o pai de Q bloqueado no modo IX ou IS.
4. Pode bloquear um nó Q no modo X (exclusivo), SIX ou IX somente se atualmente tiver o pai de Q bloqueado no modo IX ou SIX.
5. Só pode bloquear um nó se não tiver desbloqueado anteriormente qualquer nó (ou seja, T_i é em duas fases).
6. Pode desbloquear um nó Q somente se atualmente não tiver nenhum dos filhos de Q bloqueado.

Observe que o protocolo de granularidade múltipla exige que os bloqueios sejam adquiridos na ordem *de cima para baixo* (raiz-para-folha), enquanto os bloqueios precisam ser liberados na ordem *de baixo para cima* (folha-para-raiz).

Como ilustração do protocolo, considere a árvore da Figura 16.16 e estas transações:

- Suponha que a transação T_{18} leia o registro r_{a_2} no arquivo F_a. Então, T_{18} precisa bloquear o banco de dados, área A_1, e F_a no modo IS (e nessa ordem), para finalmente bloquear r_{a_2} no modo S.
- Suponha que a transação T_{19} modifique o registro r_{a_9} no arquivo F_a. Depois, T_{19} precisa bloquear o banco de dados, área A_1 e arquivo F_a no modo IX, e finalmente bloquear ra_9 no modo X.
- Suponha que a transação T_{20} leia todos os registros no arquivo F_a. Depois, T_{20} precisa bloquear o banco de dados e a área A_1 (nessa ordem) no modo IS, e finalmente bloquear F_a no modo S.
- Suponha que a transação T_{21} leia o banco de dados inteiro. Ela pode fazer isso depois de bloquear o banco de dados no modo S.

Observamos que as transações T_{18}, T_{20} e T_{21} podem acessar o banco de dados simultaneamente. A transação T_{19} pode executar simultaneamente com T_{18}, mas não com T_{20} ou T_{21}. Esse protocolo melhora a concorrência e reduz a sobrecarga de bloqueio. Ele é particularmente útil em aplicações que incluem uma mistura de

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figura 16.17 Matriz de compatibilidades.

- Transações curtas que acessam apenas alguns itens de dados
- Transações longas que produzem relatórios de um arquivo inteiro ou conjunto de arquivos

Há um protocolo de bloqueio semelhante, que se aplica aos sistemas de banco de dados em que as granularidades de dados são organizadas na forma de um gráfico acíclico direcionado. Veja as notas bibliográficas para obter referências adicionais. O impasse é possível no protocolo que temos, assim como no protocolo de bloqueio em duas fases. Existem técnicas para reduzir a freqüência de impasse no protocolo de granularidade múltiplas e também para eliminar inteiramente o impasse. Essas técnicas possuem referências nas notas bibliográficas no final do capítulo.

Esquemas de múltipla versão

Os esquemas de controle de concorrência discutidos até aqui asseguram a seriação adiando uma operação ou abortando a transação que emitiu a operação. Por exemplo, uma operação read pode ser adiada porque o valor apropriado ainda não foi escrito; ou então pode ser rejeitada (ou seja, a transação emissora precisa ser abortada) porque o valor que ela deveria ler já foi alterado. Essas dificuldades poderiam ser evitadas se cópias antigas de cada item de dados fossem mantidas em um sistema.

Nos esquemas de **controle de concorrência de múltipla versão**, cada operação `write`(Q) cria uma nova versão de Q. Quando uma transação emite uma operação `read`(Q), o gerenciador de controle de concorrência seleciona uma das versões de Q para ser lida. O esquema de controle de concorrência precisa assegurar que a versão a ser lida seja selecionada de uma maneira que assegure a seriação. Também é fundamental, por motivos de desempenho, que uma transação seja capaz de determinar fácil e rapidamente qual versão do item de dados deve ser lida.

Ordenação por timestamp em múltipla versão

A técnica de ordenação de transação mais comum utilizada por esquemas de múltipla versão é o timestamp. Com cada transação T_i no sistema, associamos um timestamp estático exclusivo, indicada por $TS(T_i)$. O sistema de banco de dados atribui essa timestamp antes que a transação inicie a execução, conforme descrito na seção "Protocolos baseados em timestamp".

Com cada item de dados Q, uma seqüência de versões $<Q_1, Q_2, ..., Q_m>$ é associada. Cada versão Q_k contém três campos de dados:

- **Conteúdo** é o valor da versão Q_k.
- **W-timestamp**(Q_k) é o timestamp da transação que criou a versão Q_k.
- **R-timestamp**(Q_k) é o maior timestamp de qualquer transação que leu com sucesso a versão Q_k.

Uma transação – digamos, T_i – cria uma nova versão Q_k do item de dados Q emitindo uma operação write(Q). O campo de conteúdo da versão mantém o valor escrito por T_i. O sistema inicializa W-timestamp e R-timestamp com $TS(T_i)$. Ele atualiza o valor de R-timestamp de Q_k sempre que uma transação T_j lê o conteúdo de Q_k, e R-timestamp(Q_k) < $TS(T_j)$.

O **esquema de ordenação por timestamp em múltipla versão** apresentado em seguida assegura a seriação. O esquema opera da seguinte forma. Suponha que a transação T_i emita uma operação read(Q) ou write(Q). Considere que Q_k indique a versão de Q cujo timestamp de escrita seja o maior timestamp de escrita menor ou igual a $TS(T_i)$.

1. Se a transação T_i emitir um read(Q), então o valor retornado é o conteúdo da versão Q_k.
2. Se a transação T_i emitir write(Q) e se $TS(T_i)$ < R-timestamp(Q_k), então o sistema reverte a transação T_i. Por outro lado, se $TS(T_i)$ = W-timestamp(Q_k), o sistema escreve sobre o conteúdo de Q_k; caso contrário, ele cria uma nova versão de Q.

A justificativa para a regra 1 é clara. Uma transação lê a versão mais recente que vem antes dela no tempo. A segunda regra força uma transação a abortar se ela estiver "muito atrasada" ao fazer uma escrita. Mais precisamente, se T_i tentar escrever uma versão que alguma outra transação teria lido, então não podemos permitir que a escrita tenha sucesso.

As versões que não são mais necessárias são removidas de acordo com a regra a seguir. Suponha que existam duas versões, Q_k e Q_j, de um item de dados, e que as duas versões tenham um W-timestamp menor que o timestamp da transação mais antiga no sistema. Então, a mais antiga das duas versões Q_k e Q_j não será usada novamente e poderá ser excluída.

O esquema de ordenação por timestamp em múltipla versão tem a propriedade desejável de que uma solicitação de leitura nunca falha e nunca precisa esperar. Em sistemas de banco de dados típicos, em que a leitura é uma operação mais freqüente do que a escrita, essa vantagem pode ser de maior significado prático.

O esquema, porém, sofre de duas propriedades indesejáveis. Primeiro, a leitura de um item de dados também exige a atualização do campo R-timestamp, resultando em dois acessos de disco em potencial, em vez de um. Segundo, os

conflitos entre transações são resolvidos por meio de rollbacks, e não esperas. Essa alternativa pode ser dispendiosa. A próxima seção descreve um algoritmo para aliviar esse problema.

Esse esquema de ordenação por timestamp em múltipla versão não assegura a facilidade de recuperação e inexistência de cascata. Ele pode ser estendido da mesma maneira que o esquema de ordenação de timestamp básico, para torná-lo recuperável e sem cascata.

Bloqueio de duas fases em múltipla versão

O protocolo de **bloqueio de duas fases em múltipla versão** tenta combinar as vantagens do controle de concorrência em múltipla versão com as vantagens do bloqueio de duas fases. Esse protocolo diferencia entre **transações somente leitura** e **transações de atualização**.

As transações de atualização realizam um bloqueio rigoroso em duas fases; ou seja, eles mantêm todos os bloqueios até o final da transação. Assim, eles podem ser seriados de acordo com sua ordem de commit. Cada versão de um item de dados possui um único timestamp. O timestamp nesse caso não é um timestamp baseado em clock, mas sim um contador, que chamaremos de ts-counter, que é incrementado durante o processamento do commit.

O sistema de banco de dados atribui um timestamp às transações somente leitura lendo o valor atual de ts-counter antes de iniciarem a execução; elas seguem o protocolo de ordenação por timestamp em múltipla versão para realizar as leituras. Assim, quando uma transação somente leitura T_i emitir um read(Q), o valor retornado é o conteúdo da versão cuja timestamp é o maior timestamp menor ou igual a $TS(T_i)$.

Quando uma transação de atualização lê um item, ela recebe um bloqueio compartilhado sobre o item e lê a versão mais recente desse item. Quando uma transação de atualização quiser escrever um item, ela primeiro apanha um bloqueio exclusivo sobre o item e depois cria uma nova versão do item de dados. A escrita é realizada na nova versão, e o timestamp da nova versão é inicialmente definida como um valor ∞, um valor maior do que o de qualquer timestamp possível.

Quando a transação de atualização T_i completar suas ações, ela executará o processamento de commit. Primeiro, T_i define o timestamp em cada versão que criou como 1 a mais do que o valor ts-counter;depois, T_i incrementa `ts-counter` em 1. Somente uma transação de atualização tem permissão para realizar o processamento de commit de cada vez.

Como resultado, as transações somente leitura que iniciam depois que T_i incrementa ts-counter verão os valores atualizados por T_i,enquanto aquelas que iniciam antes que T_i incremente ts-counter verão o valor antes das atualizações por T_i. Em ambos os casos, as transações somente leitura nunca precisam esperar pelos bloqueios. O bloqueio de duas fases em múltipla versão também garante que os schedules são recuperáveis e sem cascata.

As versões são excluídas de maneira semelhante à da ordenação por timestamp em múltipla versão. Suponha que existam duas versões, Q_k e Q_j, de um item de dados, e que as duas versões tenham um timestamp menor ou igual ao timestamp da transação somente leitura mais antiga no sistema. Depois, a mais antiga das duas versões Q_k e Q_j não será usada novamente e poderá ser excluída.

O bloqueio de duas fases em múltipla versão, ou suas variações, é usado em alguns sistemas de banco de dados comerciais.

Tratamento de impasse

Um sistema está em um estado de impasse se houver um conjunto de transações tal que cada transação no conjunto está esperando por outra transação no conjunto. Mais precisamente, existe um conjunto de transações aguardando $\{T_0, T_1, ..., T_n\}$ tal que T_0 está esperando por um item de dados que T_1 mantém, e T_1 está esperando por um item de dados que T_2 mantém, ..., e T_{n-1} está esperando por um item de dados que T_n mantém, e T_n está esperando por um item de dados que T_0 mantém. Nenhuma das transações pode prosseguir em tal situação.

A única solução para essa situação indesejável é que o sistema invoque alguma ação drástica, como reverter algumas das transações envolvidas no impasse. O rollback de uma transação pode ser parcial. Ou seja, uma transação pode ser revertida ao ponto em que obteve um bloqueio cuja liberação resolve o impasse.

Existem dois métodos especiais para lidar com o problema do impasse. Podemos usar um protocolo de **prevenção de impasse** para garantir que o sistema *nunca* entrará em um estado de impasse. Como alternativa, podemos permitir que o sistema entre em um estado de impasse e depois tente recuperar, usando um esquema de **detecção de impasse e recuperação de impasse**. Como veremos, os dois métodos podem resultar em um rollback da transação. A prevenção normalmente é usada se a probabilidade de o sistema entrar em um estado de impasse for relativamente alta; caso contrário, a detecção e a recuperação são mais eficientes.

Observe que um esquema de detecção e recuperação exige uma sobrecarga que inclui não apenas o custo em tempo de execução de manter a informação necessária e executar o algoritmo de detecção, mas também as perdas em potencial inerentes à recuperação de um impasse.

Prevenção de impasse

Existem duas técnicas para prevenção de impasse. Uma técnica assegura que nenhuma espera cíclica poderá ocorrer pela ordenação das solicitações para bloqueios, ou pela exigência de que todos os bloqueios sejam adquiridos juntos. A outra técnica é mais próxima da recuperação de impasse, e realiza o rollback da transação em vez de esperar por um impasse, sempre que a espera puder potencialmente resultar em um impasse.

O esquema mais simples sob a primeira técnica exige que cada transação bloqueie todos os seus itens de dados antes de iniciar a execução. Além do mais, ou todos são bloqueados em uma etapa ou nenhum é bloqueado. Existem duas desvantagens importantes para esse protocolo: (1) normalmente, é difícil prever, antes que a transação seja iniciada, que itens de dados precisam ser bloqueados; (2) a utilização do item de dados pode ser muito baixa, pois muitos dos itens de dados podem ser bloqueados, mas não utilizados por um longo tempo.

Outra técnica para impedir impasses é impor uma ordenação de todos os itens de dados e exigir que uma transação bloqueie itens de dados somente em uma seqüência consistente com a ordenação. Vimos um esquema desse tipo no protocolo de árvore, que usa uma ordenação parcial dos itens de dados.

Uma variação dessa técnica é usar uma ordem total dos itens de dados, em conjunto com o bloqueio em duas fases. Quando uma transação tiver bloqueado um item em particular, ela não poderá solicitar bloqueios sobre itens que precedem esse item na ordenação. Esse esquema é fácil de implementar, desde que o conjunto de itens de dados acessados por uma transação seja conhecido quando a transação iniciar a execução. Não é preciso mudar o sistema básico de controle de concorrência se o bloqueio em duas fases for utilizado. Basta assegurar que os bloqueios sejam solicitados na ordem correta.

A segunda técnica para impedir impasses é usar preempção e rollbacks de transação. Na preempção (ou apropriação), quando uma transação T_2 solicita um bloqueio que a transação T_1 mantém, o bloqueio concedido a T_1 pode ser **apropriado** pela reversão de T_1 e concessão do bloqueio a T_2. Para controlar a preempção, atribuímos um timestamp exclusivo a cada transação. O sistema usa esses timestamps somente para decidir se uma transação deverá esperar ou reverter. O bloqueio ainda é usado para o controle de concorrência. Se uma transação for revertida, ela retém seu timestamp *antigo* quando reiniciada. Dois esquemas diferentes de prevenção de impasse usando timestamps foram propostos:

1. O esquema **esperar-morrer** é uma técnica não preemptiva. Quando a transação T_i solicita um item de dados atualmente mantido por T_j, T_i tem permissão para esperar somente se tiver um timestamp menor do que o de T_j (ou seja, T_i é mais velho que T_j). Caso contrário, T_i é revertido (morre).

   Por exemplo, suponha que as transações T_{22}, T_{23} e T_{24} possuem timestamps 5, 10 e 15, respectivamente. Se T_{22} solicitar um item de dados mantido por T_{23}, então T_{22} esperará. Se T_{24} solicitar um item de dados mantido por T_{23}, então T_{24} será revertido.
2. O esquema **ferir-esperar** é uma técnica preemptiva. Esse é um complemento ao esquema esperar-morrer. Quando a transação T_i solicita um item de dados atualmente mantido por T_j, T_i tem permissão para esperar somente se tiver um timestamp maior que o de T_j (ou seja, T_i é mais novo que T_j). Caso contrário, T_j é revertido (T_j é *ferido* por T_i).

   Retornando ao nosso exemplo, com as instruções T_{22}, T_{23} e T_{24}, se T_{22} solicitar um item de dados mantido por T_{23}, então o item de dados será apropriado de T_{23}, e T_{23} será revertido. Se T_{24} solicitar um item de dados mantido por T_{23}, então T_{24} esperará.

Sempre que o sistema reverter transações, é importante garantir que não haja **estagnação** – ou seja, nenhuma transação será revertida repetidamente, sem obter permissão para prosseguir.

Os esquemas ferir-esperar e esperar-morrer evitam a estagnação. A qualquer momento, haverá uma transação com o menor timestamp. *Não se pode* exigir que essa transação reverta em ambos os esquemas. Como os timestamps sempre aumentam, e como as transações *não* recebem novos timestamps quando são revertidas, uma transação que é revertida repetidamente por fim terá o menor timestamp, e nesse ponto ela não será mais revertida.

Porém, existem diferenças significativas no modo como os dois esquemas operam.

- No esquema esperar-morrer, uma transação mais antiga precisa esperar que uma mais nova libere seu item de dados. Assim, quanto mais velha uma transação fica, mais ela tende a esperar. Ao contrário, no esquema ferir-esperar, uma transação mais antiga nunca espera por uma transação mais nova.
- No esquema esperar-morrer, se uma transação T_i morrer e for revertida porque solicitou um item de dados mantido pela transação T_j, então T_i pode emitir a mesma seqüência de solicitações novamente quando for reiniciada. Se o item de dados ainda for mantido por T_j, então T_i morrerá novamente. Assim, T_i poderá morrer várias vezes antes de adquirir o item de dados necessário. Compare essa série de eventos com o que acontece no esquema ferir-esperar. A transação T_i é ferida e revertida porque T_j solicitou um item de dados que ela mantém.

Quando T_i é reiniciada e solicita um item de dados agora sendo mantido por T_j, T_i espera. Assim, pode haver menos rollbacks no esquema ferir-esperar.

O maior problema com esses dois esquemas é que podem ocorrer rollbacks desnecessários.

Esquemas baseados em tempo limite

Outra técnica simples para o tratamento do impasse é baseada nos **tempos limite de bloqueio**. Nessa técnica, uma transação que solicitou um bloqueio espera por no máximo uma quantidade de tempo especificada. Se o bloqueio não tiver sido concedido dentro desse tempo, diz-se que a transação esgotou seu tempo limite, e se reverte e reinicia. Se houvesse de fato um impasse, uma ou mais transações envolvidas no impasse esgotarão o tempo limite e reverterão, permitindo que outras prossigam. Esse esquema se situa em algum lugar entre a prevenção de impasse, em que um impasse nunca ocorrerá, e a detecção e recuperação de impasse, que são discutidos na próxima seção.

O esquema de tempo limite é particularmente fácil de implementar, e funciona bem se as transações forem curtas e se longas esperas provavelmente forem devidas a impasses. Porém, em geral, é difícil decidir por quanto tempo uma transação precisa esperar antes de esgotar o tempo limite. Uma espera muito longa resulta em atrasos desnecessários quando um impasse tiver ocorrido. Uma espera muito curta resulta em rollback de transação mesmo quando não existe impasse, levando a recursos desperdiçados. A estagnação também é uma possibilidade com esse esquema. Logo, o esquema baseado em tempo limite possui aplicação limitada.

Detecção e recuperação de impasse

Se um sistema não empregar algum protocolo que assegure a liberdade de impasse, então um esquema de detecção e recuperação precisa ser usado. Um algoritmo que examina o estado do sistema é invocado periodicamente para determinar se um impasse ocorreu. Se tiver ocorrido, então o sistema precisa tentar se recuperar dele. Para isso, o sistema precisa:

- Manter informações sobre a alocação atual dos itens de dados para transações, além de quaisquer solicitações de itens de dados pendentes.
- Oferecer um algoritmo que use essa informação para determinar se o sistema entrou em um estado de impasse.
- Recuperar-se de impasse quando o algoritmo de detecção determinar que existe um impasse. Nesta seção, discutimos essas questões.

Detecção de impasse

Impasses podem ser descritos com precisão em termos de um grafico direcionado, chamado gráfico de espera (*wait for*). Esse gráfico consiste em um par $G = (V, E)$, em que V é um conjunto de vértices e E é um conjunto de arestas. O conjunto de vértices consiste em todas as transações no sistema. Cada elemento no conjunto E de arestas é um par ordenado $T_i \rightarrow T_j$. Se $T_i \rightarrow T_j$ estiver em E, então existe uma aresta direcionada de transações T_i para T_j, implicando que a transação T_i está esperando até que a transação T_j libere um item de dados de que precisa.

Quando a transação T_i solicita um item de dados atualmente sendo mantido pela transação T_j, então a aresta $T_i \rightarrow T_j$ é inserida no gráfico de espera. Essa aresta só é removida quando a transação T_j não está mais mantendo um item de dados necessário pela transação T_i.

Existe um impasse no sistema se e somente se o gráfico de espera contém um ciclo. Cada transação envolvida no ciclo é considerada como estando em impasse. Para detectar impasses, o sistema precisa manter o gráfico de espera e periodicamente invocar um algoritmo que procura um ciclo no gráfico.

Para ilustrar esses conceitos, considere o gráfico de espera na Figura 16.18, que representa a seguinte situação:

- A transação T_{25} está esperando pelas transações T_{26} e T_{27}.
- A transação T_{27} está esperando pela transação T_{26}.
- A transação T_{26} está esperando pela transação T_{28}.

Como o gráfico não possui ciclo, o sistema não está em um estado de impasse.

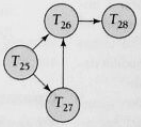


Figura 16.18 Gráfico de espera sem ciclo.

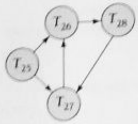


Figura 16.19 Gráfico de espera com um ciclo.

Suponha agora que a transação T_{28} esteja solicitando um item mantido por T_{27}. A aresta $T_{28} \rightarrow T_{27}$ é acrescentada ao gráfico de espera, resultando no novo estado do sistema da Figura 16.19. Desta vez, o gráfico contém o ciclo

$$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$$

implicando que as transações T_{26}, T_{27} e T_{28} estão todas em impasse.

Conseqüentemente, surge a pergunta: quando devemos invocar o algoritmo de detecção? A resposta depende de dois fatores:

1. Com que freqüência ocorre um impasse?
2. Quantas transações serão afetadas pelo impasse?

Se os impasses ocorrem com freqüência, então o algoritmo de detecção deve ser invocado com mais freqüência do que o normal. Os itens de dados alocados a transações em impasse estarão indisponíveis a outras transações até que o impasse possa ser quebrado. Além disso, o número de ciclos no gráfico também pode crescer. No pior caso, invocaríamos o algoritmo de detecção toda vez que uma solicitação de alocação não pudesse ser concedida imediatamente.

Recuperação de impasse

Quando um algoritmo de detecção determina que existe um impasse, o sistema precisa **recuperar-se** dele. A solução mais comum é reverter uma ou mais transações para quebrar o impasse. Três ações precisam ser tomadas:

1. **Seleção de uma vítima**. Dado um conjunto de transações em impasse, temos de determinar qual transação (ou transações) reverter para quebrar o impasse. É preciso reverter aquelas transações que incorrerão em custo mínimo. Infelizmente, o termo *custo mínimo* não é exato. Muitos fatores podem determinar o custo de um rollback, incluindo
   a. Por quanto tempo a transação executou e quanto mais tempo a transação continuará executando até concluir sua tarefa designada.
   b. Quantos itens de dados a transação usou.
   c. Quantos mais itens de dados a transação precisa para completar.
   d. Quantas transações estarão envolvidas no rollback.
2. **Rollback**. Quando tivermos decidido que uma transação em particular precisará ser revertida, temos de determinar até que ponto ela deverá ser revertida.

   A solução mais simples é um **rollback total**. Aborte a transação e depois reinicie-a. Porém, é mais eficiente reverter a transação apenas até o ponto necessário para quebrar o impasse. Esse **rollback parcial** exige que o sistema mantenha informações adicionais sobre o estado de todas as transações em execução. Especificamente, a seqüência de solicitações/concessões de bloqueio e atualizações realizadas pela transação precisa ser registrada. O mecanismo de detecção de impasse deverá decidir quais bloqueios a transação selecionada precisa liberar a fim de quebrar o impasse. A transação selecionada precisa ser revertida até o ponto em que obteve o primeiro desses bloqueios, desfazendo todas as ações tomadas após esse ponto. O mecanismo de recuperação precisa ser capaz de realizar esses rollbacks parciais. Além do mais, as transações precisam ser capazes de retomar a execução após um rollback parcial. Veja as notas bibliográficas para obter referências relevantes.
3. **Estagnação**. Em um sistema em que a seleção de vítimas é baseada principalmente em fatores de custo, pode acontecer que a mesma transação sempre seja apanhada como uma vítima. Como resultado, essa transação nunca termina sua tarefa designada, daí a **estagnação**. Temos de garantir que uma transação poderá ser apanhada como uma vítima apenas por um número finito (e pequeno) de vezes. A solução mais comum é incluir o número de rollbacks no fator de custo.

Operações de inserção e exclusão

Até agora, restringimos nossa atenção para operações read e write. Essa restrição limita as transações a itens de dados

já no banco de dados. Algumas transações exigem não apenas acesso a itens de dados existentes, mas também a capacidade de criar novos itens de dados. Outras exigem a capacidade de excluir itens de dados. Para examinar como essas transações afetam o controle de concorrência, introduzimos estas operações adicionais:

- delete(Q) exclui o item de dados Q do banco de dados.
- insert(Q) insere um novo item de dados Q no banco de dados e atribui a Q um valor inicial.

Uma tentativa por uma transação T_i de realizar uma operação read(Q) após Q ter sido excluído resulta em um erro lógico em T_i. De modo semelhante, uma tentativa por uma transação T_i de realizar uma operação read(Q) antes que Q tenha sido inserido resulta em um erro lógico em T_i. Também é um erro lógico tentar excluir um item de dados inexistente.

Exclusão

Para entender como a presença de instruções delete afeta o controle de concorrência, temos de decidir quando uma instrução **delete** entra em conflito com outra instrução. Considere que I_i e I_j sejam instruções de T_i e T_j, respectivamente, que aparecem no schedule S em ordem consecutiva. Considere que I_i = **delete**(Q). Consideramos várias instruções I_j.

- I_j = read(Q). I_i e I_j entram em conflito. Se I_i vier antes de I_j, T_j terá um erro lógico. Se I_j vier antes de I_i, T_j poderá executar a operação read com sucesso.
- Ij = write(Q). Ii e Ij entram em conflito. Se Ii vier antes de Ij, Tj terá um erro lógico. Se Ij vier antes de Ii, Tj poderá executar a operação write com sucesso.
- I_j = **delete**(Q). I_i e I_j entram em conflito. Se I_i vier antes de I_j, T_i terá um erro lógico. Se I_j vier antes de I_i, T_i terá um erro lógico.
- Ij = insert(Q). Ii e Ij entram em conflito. Suponha que o item de dados Q não existisse antes da execução de I_i e I_j. Então, se I_i vier antes de I_j, um erro lógico resultará para T_i. Se I_j vier antes de I_i, então nenhum erro lógico resultará. De modo semelhante, se Q existisse antes da execução de I_i e I_j, então um erro lógico resultará se I_j vier antes de I_i, mas não de outra forma.

Podemos concluir o seguinte:

- Sob o protocolo de bloqueio em duas fases, um bloqueio exclusivo é exigido sobre um item de dados antes que esse item possa ser excluído.
- Sob o protocolo de ordenação por timestamp, um teste semelhante ao de um write precisa ser realizado. Suponha que a transação T_i emita **delete**(Q).
  - Se $TS(T_i) < R\text{-timestamp}(Q)$, então o valor de Q que T_i estava para excluir já foi lido por uma transação T_j com $TS(T_j) > TS(T_i)$. Logo, a operação **delete** é rejeitada e T_i é revertida.
  - Se $TS(T_i) < W\text{-timestamp}(Q)$, então uma transação T_j com $TS(T_j) > TS(T_i)$ escreveu Q. Logo, essa operação **delete** é rejeitada, e T_i é revertida.
  - Caso contrário, o **delete** é executado.

Inserção

Já vimos que uma operação insert(Q) entra em conflito com uma operação **delete**(Q). De modo semelhante, **insert**(Q) entra em conflito com uma operação read(Q) ou uma operação **write** (Q); nenhum read ou write pode ser realizado sobre um item de dados antes que ele exista.

Como um **insert**(Q) atribui um valor ao item de dados Q, um **insert** é tratado de modo semelhante a um write para fins de controle de concorrência:

- Sob o protocolo de bloqueio em duas fases, se T_i realizar uma operação **insert**(Q), T_i receberá um bloqueio exclusivo sobre o item de dados Q recém-criado.
- Sob o protocolo de ordenação de timestamp, se T_i realizar uma operação **insert**(Q), os valores R-timestamp(Q) e W-timestamp(Q) são definidos como $TS(T_i)$.

O fenômeno fantasma

Considere a transação T29, que executa a seguinte consulta SQL sobre o banco de dados bancário:

```
select sum(saldo)
from conta
where nome_agência = 'Perryridge'
```

A tr T_{29} requer acesso a todas as tuplas da relação *conta* pertencentes à agência Perryridge. Considere que $T_{-(30)}$ seja uma transação que executa a seguinte inserção SQL:

```
insert into conta
    values (A-201, 'Perryridge', 900)
```

Considere que S seja um schedule envolvendo T_{29} e T_{30}. Esperamos que haja um potencial para conflito pelos seguintes motivos:

- Se T_{29} utiliza a tupla recém-inserida por T_{30} no cálculo de **sum**(*saldo*), então T_{29} lê um valor escrito por T_{30}. Assim, em um schedule serial equivalente a S, T_{30} precisa vir antes de T_{29}.
- Se T_{29} não usar a tupla recém-inserida por T_{30} no cálculo de **sum**(*saldo*), então, em um schedule serial equivalente a S, T_{29} precisa vir antes de T_{30}.

O segundo desses dois casos é curioso. T_{29} e T_{30} não acessam qualquer tupla em comum, mas entram em conflito um com o outro! Com efeito, T_{29} e T_{30} entram em conflito em uma tupla fantasma. Se o controle de concorrência for realizado na granularidade de tupla, esse conflito passaria sem ser detectado. Como resultado, o sistema poderia falhar para impedir um schedule não passível de seriação. Esse problema é chamado de **fenômeno fantasma**.

Para impedir o fenômeno fantasma, permitimos que a transação T_{29} impeça outras transações de criar novas tuplas na relação *conta* com *nomeagência* = "Perryridge".

Para encontrar todas as tuplas *conta* com *nomeagência* = "Perryridge", T_{29} precisa pesquisar a relação *conta* inteira, ou pelo menos um índice sobre a relação. Até agora, consideramos implicitamente que os únicos itens de dados acessados por uma transação são tuplas. Porém, T_{29} é um exemplo de uma transação que lê informações sobre que tuplas estão em uma relação, e T_{30} é um exemplo de uma transação que atualiza essa informação.

Claramente, não é suficiente apenas bloquear as tuplas que são acessadas; a informação usada para encontrar as tuplas acessadas pela transação também precisa ser bloqueada.

A solução mais simples para esse problema é associar um item de dados à relação; o item de dados representa a informação usada para encontrar as tuplas na relação. As transações, como T_{29}, que lêem a informação sobre quais tuplas estão em uma relação, teriam então de bloquear o item de dados correspondente à relação no modo compartilhado. As transações, como T_{30}, que atualizam as informações sobre quais tuplas estão em uma relação, teriam de bloquear o item de dados no modo exclusivo. Assim, T_{29} e T_{30} estariam em conflito em um item de dados real, em vez de um fantasma.

Não confunda o bloqueio de uma relação inteira, como no bloqueio de granularidade múltipla, com o bloqueio do item de dados correspondente à relação. Bloqueando o item de dados, uma transação só impede que outras transações atualizem as informações sobre quais tuplas estão na relação. O bloqueio ainda é exigido sobre tuplas. Uma transação que acessa diretamente uma tupla pode receber um bloqueio sobre as tuplas mesmo quando outras transações possuem um bloqueio exclusivo sobre o item de dados correspondente à própria relação.

A principal desvantagem de bloquear um item de dados correspondente à relação é o baixo grau de concorrência – duas transações que inserem diferentes tuplas em uma relação são impedidas de executar simultaneamente.

Uma solução melhor é a técnica de **bloqueio de índice**. Qualquer transação que insere uma tupla em uma relação precisa inserir informações em cada índice mantido na relação. Eliminamos o fenômeno fantasma impondo um protocolo de bloqueio para índices. Para simplificar, consideraremos apenas índices de árvore B+.

Como vimos no Capítulo 12, cada valor de chave de busca está associado a um nó de folha de índice. Uma consulta normalmente usará um ou mais índices para acessar uma relação. Uma inserção precisa inserir a nova tupla em todos os índices na relação. Em nosso exemplo, consideramos que existe um índice sobre *conta* para *nome_agência*. Depois, T_{30} precisa modificar a folha contendo a chave Perryridge. Se T_{29} ler o mesmo nó de folha para localizar todas as tuplas pertencentes à agência Perryridge, então T_{29} e T_{30} entram em conflito nesse nó de folha.

O **protocolo de bloqueio de índice** tira proveito da disponibilidade de índices em uma relação, transformando casos de fenômeno fantasma em conflitos sobre bloqueios nos nós folha de índice. O protocolo opera da seguinte forma:

- Cada relação precisa ter pelo menos um índice.
- Uma transação T_i só pode acessar tuplas de uma relação depois de primeiro encontrá-las por um ou mais dos índices na relação.
- Uma transação T_i que realiza uma pesquisa (seja uma pesquisa de intervalo ou uma pesquisa pontual) precisa adquirir um bloqueio compartilhado sobre todos os nós de folha de índice que ela acessa.
- Uma transação T_i não pode inserir, excluir ou atualizar uma tupla t_i em uma relação r sem atualizar todos os índices sobre r. A transação precisa obter bloqueios exclusivos sobre todos os nós de folha que são afetados pela inserção, exclusão ou atualização. Para inserção e exclusão, os nós de folha afetados são aqueles que contêm (após a inserção) ou continham (antes da exclusão) o valor da chave de busca da tupla. Para as atualizações, os nós de folha afetados são aqueles que (antes da modificação) continham o valor antigo da chave de busca e os nós que (após a modificação) contêm o novo valor da chave de busca.
- As regras do protocolo de bloqueio em duas fases precisam ser observadas.

As variantes da técnica de bloqueio de índice existem para eliminar o fenômeno fantasma sob os outros protocolos de controle de concorrência apresentados neste capítulo.

Níveis de consistência fracos

A seriação é um conceito útil, pois permite que os programadores ignorem questões relacionadas a concorrência quando codificam transações. Se cada transação tiver a propriedade de manter a consistência do banco de dados se executada sozinha, então a seriação assegura que as execuções simultâneas mantêm a consistência. Porém, os protocolos exigidos para garantir a seriação podem permitir muito pouca concorrência para certas aplicações. Nesses

casos, níveis de consistência mais fracos são utilizados. O uso de níveis de consistência mais fracos impõe um peso adicional sobre os programadores, para garantir a exatidão do banco de dados.

Consistência de grau dois

A finalidade da **consistência** de grau dois é evitar os abortos em cascata sem necessariamente assegurar a seriação. O protocolo de bloqueio para a consistência de grau dois utiliza os mesmos dois modos de bloqueio que usamos para o protocolo de bloqueio de duas fases: compartilhado (S) e exclusivo (X). Uma transação precisa manter o modo de bloqueio apropriado quando acessar um item de dados.

Ao contrário da situação no bloqueio de duas fases, os bloqueios S podem ser liberados um de cada vez, e os bloqueios podem ser adquiridos um de cada vez. Os bloqueios exclusivos não podem ser liberados até que a transação submeta ou aborte. A seriação não é garantida por esse protocolo. Na realidade, uma transação pode ler o mesmo item de dados duas vezes e obter diferentes resultados. Na Figura 16.20, T_3 lê o valor de Q antes e depois que esse valor é escrito por T_4.

O potencial para incoerência devido a schedules não passíveis de seriação sobre a consistência de grau dois torna essa técnica indesejável para muitas aplicações.

Estabilidade de cursor

A **estabilidade de cursor** é uma forma de consistência de grau dois criada para programas escritos nas linguagens host, que percorrem as tuplas de uma relação usando cursores. Em vez de bloquear a relação inteira, a estabilidade do cursor garante que

- A tupla que atualmente está sendo processada pela iteração seja bloqueada no modo compartilhado.
- Quaisquer tuplas modificadas sejam bloqueadas no modo exclusivo até que a transação seja confirmada.

Essas regras garantem que a consistência de grau dois seja obtida. O bloqueio em duas fases não é exigido. A seriação não é garantida. A estabilidade do cursor é usada na prática sobre relações muito acessadas como um meio de aumentar a concorrência e melhorar o desempenho do sistema. As aplicações que usam a estabilidade do cursor precisam ser codificadas de um modo que garanta a consistência do banco de dados apesar da possibilidade de schedules não passíveis de seriação. Assim, o uso da estabilidade do cursor é limitado a situações especializadas com restrições de coerência simples.

Níveis de consistência fracos em SQL

O padrão SQL também permite que uma transação especifique que ela pode ser executada de modo que se torne não passível de seriação com relação a outras transações. Por exemplo, uma transação pode operar no nível de **leitura não confirmada**, que permite que a transação leia registros mesmo que eles não tenham sido confirmados. A SQL oferece tais recursos para transações longas cujos resultados não precisam ser exatos. Por exemplo, em geral, a informação aproximada é suficiente para estatísticas usadas na otimização da consulta. Se essas transações tivessem de ser executadas em um padrão passível de seriação, elas poderiam interferir com outras transações, causando o adiamento da execução das outras.

Os níveis de consistência especificados pela SQL-92 são os seguintes:

- **Passível de seriação** é o default.
- **Leitura repetitiva** permite que somente registros confirmados sejam lidos e ainda exige que, entre duas leituras de um registro por uma transação, nenhuma outra transação tenha permissão para atualizar o registro. Porém, a transação pode não ser passível de seriação com relação a outras transações. Por exemplo, quando está procurando registros que satisfazem algumas condições,

T_3	T_4
lock-S(Q)	
read(Q)	
unlock(Q)	
	lock-X(Q)
	read(Q)
	write(Q)
	unlock(Q)
lock-S(Q)	
read(Q)	
unlock(Q)	

Figura 16.20 Schedule não passível de seriação com consistência de grau dois.

uma transação pode encontrar alguns dos registros inseridos por uma transação confirmada, mas pode não encontrar outros.

- **Leitura confirmada** permite que somente registros confirmados sejam lidos, mas não exige sequer leituras repetitivas. Por exemplo, entre duas leituras de um registro pela transação, os registros podem ter sido atualizados por outras transações confirmadas. Isso é basicamente o mesmo que consistência de grau dois; a maioria dos sistemas admitindo esse nível de consistência realmente implementaria a estabilidade de cursor, que é um caso especial de consistência de grau dois.
- **Leitura não confirmada** permite até mesmo que registros não confirmados sejam lidos. Esse é o nível mais baixo de coerência permitido pela SQL-92.

Concorrência em estruturas de índice**

É possível tratar o acesso a estruturas de índice como qualquer outra estrutura de banco de dados e aplicar as técnicas de controle de concorrência discutidas anteriormente. Porém, como os índices são acessados com freqüência, eles se tornariam um ponto de grande disputa de bloqueio, levando a um baixo grau de concorrência. Felizmente, os índices não precisam ser tratados como outras estruturas de banco de dados. É perfeitamente aceitável que uma transação realize uma pesquisa sobre um índice duas vezes e descubra que a estrutura do índice mudou nesse interim, desde que a pesquisa de índice retorne o conjunto correto de tuplas. Assim, é aceitável ter acesso simultâneo não passível de seriação a um índice, desde que a precisão do índice seja mantida.

Esboçamos duas técnicas para gerenciar o acesso concorrente às árvores B+. As notas bibliográficas referenciam outras técnicas para árvores B+, bem como técnicas para outras estruturas de índice.

As técnicas que apresentamos para controle de concorrência sobre árvores B+ são baseadas no bloqueio, mas nem o bloqueio de duas fases nem o protocolo de árvore é empregado. Os algoritmos para pesquisa, inserção e exclusão são aqueles usados no Capítulo 12, com apenas algumas modificações.

A primeira técnica é chamada de **protocolo caranguejo**:

- Ao procurar um valor de chave, o protocolo caranguejo primeiro bloqueia o nó raiz no modo compartilhado. Ao atravessar a árvore, ele adquire um bloqueio compartilhado sobre o nó filho a ser atravessado mais tarde. Depois de adquirir o bloqueio sobre o nó filho, ele libera o bloqueio sobre o nó pai e repete esse processo até que alcance o nó de folha.
- Ao inserir ou excluir um valor de chave, o protocolo caranguejo toma estas ações:
  - Ele segue o mesmo protocolo da pesquisa até alcançar o nó de folha desejado. Até esse ponto, ele obtém (e libera) apenas bloqueios compartilhados.
  - Ele bloqueia o nó de folha no modo exclusivo e insere ou exclui o valor de chave.
  - Se precisar dividir um nó ou uni-lo aos seus irmãos, ou redistribuir os valores de chave entre os irmãos, o protocolo caranguejo bloqueia o pai do nó no modo exclusivo. Depois de realizar essas ações, ele libera os bloqueios sobre o nó e irmãos.

  Se o pai exigir divisão, união ou redistribuição dos valores de chave, o protocolo reterá o bloqueio sobre o pai e a divisão, união ou redistribuição se propaga ainda mais da mesma maneira. Caso contrário, ele libera o bloqueio sobre o pai.

O protocolo recebe esse nome pelo modo como os caranguejos se movem de lado, movendo as pernas em um lado, depois as pernas no outro, e continuando alternadamente. O progresso do bloqueio enquanto o protocolo desce na árvore e sobe de volta (no caso de divisão, união ou redistribuição) prossegue de maneira semelhante, tipo caranguejo.

Quando uma operação em particular libera um bloqueio sobre um nó, outras operações podem acessar esse nó. Existe uma possibilidade de impasses entre as operações de busca descendo da árvore, e divisão, união ou redistribuição propagando-se para cima na árvore. O sistema pode facilmente lidar com tais impasses reiniciando a operação de busca pela raiz, depois de liberar os bloqueios mantidos pela operação.

A segunda técnica alcança ainda mais concorrência, evitando até mesmo manter o bloqueio sobre um nó enquanto adquire o bloqueio sobre outro nó, usando uma versão modificada das árvores B+ chamada **árvores B-link**; as árvores B-link exigem que cada nó (incluindo nós internos, não apenas as folhas) mantenha um ponteiro para o seu irmão da direita. Esse ponteiro é exigido porque uma pesquisa que ocorre enquanto um nó está sendo dividido pode ter de pesquisar não apenas esse nó, mas também o irmão da direita desse nó (se existir). Ilustraremos essa técnica com um exemplo mais tarde, mas primeiro apresentamos os procedimentos modificados do **protocolo de bloqueio de árvore B-link**.

- **Pesquisa**. Cada nó da árvore B+ precisa ser bloqueado no modo compartilhado antes de ser acessado. Um bloqueio sobre um nó não de folha é liberado antes que qualquer bloqueio sobre qualquer outro nó na árvore B+ seja solicitado. Se uma divisão ocorrer simultaneamente com uma pesquisa, o valor de chave de

busca desejado pode não aparecer mais dentro do intervalo de valores representados por um nó acessado durante a pesquisa. Nesse caso, o valor da chave de busca está no intervalo representado por um nó irmão, que o sistema localiza seguindo o ponteiro ao irmão da direita. Porém, o sistema bloqueia os nós de folha seguindo o protocolo de bloqueio em duas fases, como a seção "O fenômeno fantasma" descreve, para evitar o fenômeno fantasma.

- **Inserção e exclusão.** O sistema segue as regras de pesquisa para localizar o nó de folha em que fará a inserção ou exclusão. Ele atualiza o bloqueio no modo compartilhado sobre esse nó para o modo exclusivo, e realiza a inserção ou exclusão. Ele bloqueia nós de folha afetados pela inserção ou exclusão seguindo o protocolo de bloqueio em duas fases, como descreve a seção "O fenômeno fantasma", para evitar o fenômeno fantasma.
- **Divisão.** Se a transação dividir um nó, ela cria um novo nó de acordo com o algoritmo da seção "Arquivos de índice de árvore B+" do Capítulo 12 e o torna o irmão da direita do nó original. Os ponteiros do irmão direito do nó original e do novo nó são definidos. Depois disso, a transação libera o bloqueio exclusivo sobre o nó original (desde que seja um nó interno; os nós de folha são bloqueados pelo modo de duas fases) e depois solicita um bloqueio exclusivo sobre o pai, para que possa inserir um ponteiro para o novo nó. (Não há necessidade de bloquear ou desbloquear o novo nó.)
- **União.** Se um nó tiver muito poucos valores de chave de busca após uma exclusão, o nó com o qual será unido terá de ser bloqueado no modo exclusivo. Quando a transação tiver unido esses dois nós, ela solicitará um bloqueio exclusivo sobre o pai, para que o nó excluído possa ser removido. Nesse ponto, a transação libera os bloqueios sobre os nós unidos. A menos que o nó pai também tenha de ser unido, seu bloqueio é liberado. Observe esse fato importante: uma inserção ou exclusão pode bloquear um nó, desbloqueá-lo e mais tarde bloqueá-lo novamente. Além do mais, uma pesquisa executada simultaneamente com uma operação de divisão ou união poderá descobrir que a chave de busca desejada foi movida para o nó irmão da direita pela operação de divisão ou união.

Como ilustração, considere a árvore B+ na Figura 16.21. Considere que existem duas operações simultâneas nessa árvore B+:

1. Inserir "Clearview"
2. Pesquisar "Downtown"

Vamos supor que a operação de inserção comece primeiro. Ela faz uma pesquisa sobre "Clearview" e descobre que o nó em que "Clearview" deve ser inserido está cheio. Portanto, ela converte seu bloqueio compartilhado sobre o nó para o modo exclusivo, e cria um novo nó. O nó original agora contém os valores de chave de busca "Brighton" e "Clearview". O novo nó contém o valor de chave de busca "Downtown".

Agora suponha que ocorra uma troca de contexto, resultando no controle passando para a operação de pesquisa. Essa operação de pesquisa acessa a raiz e segue o ponteiro para o filho esquerdo da raiz. Depois, ela acessa aquele nó e obtém um ponteiro para o filho esquerdo. Esse nó filho esquerdo originalmente continha os valores de chave de busca "Brighton" e "Downtown". Como ele atualmente é bloqueado pela operação de inserção no modo exclusivo, a operação de pesquisa precisa esperar. Observe que, nesse ponto, a operação de pesquisa não mantém qualquer bloqueio!

A operação de inserção agora desbloqueia o nó de folha e bloqueia novamente seu pai, dessa vez no modo exclusivo. Ela completa a inserção, deixando a árvore B+ como na Figura 16.22. A operação de pesquisa prossegue. Porém, ela está mantendo um ponteiro para um nó de folha incorreto. Portanto, ela segue o ponteiro irmão da direita para localizar o próximo nó. Se esse nó também estiver incorreto, a pesquisa segue o ponteiro do irmão da direita desse nó. Pode-se mostrar que, se uma pesquisa mantiver um ponteiro para um nó incorreto, então, seguindo ponteiros do irmão da direita, a pesquisa precisa por fim alcançar o nó correto.

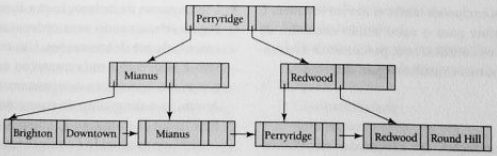


Figura 16.21 Árvore B+ para arquivo *conta* com $n = 3$.

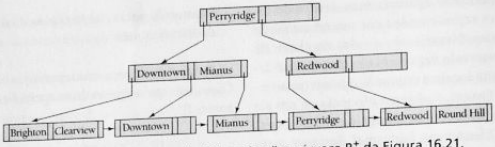


Figura 16.22 Inserção de "Clearview" na árvore B^+ da Figura 16.21.

Operações de pesquisa e inserção não podem levar a impasse. A união de nós durante a exclusão pode causar inconsistências, pois uma pesquisa pode ter lido um ponteiro para um nó excluído a partir de seu pai, antes que o nó pai fosse atualizado, e pode então tentar acessar o nó excluído. A pesquisa, então, teria de reiniciar pela raiz. Deixar nós não unidos evita essas inconsistências. Essa solução resulta em nós que contêm muito poucos valores de chave de busca e que infringem algumas propriedades das árvores B+. Porém, na maioria dos bancos de dados, as inserções são mais frequentes do que as exclusões, de modo que é provável que os nós que possuem muito poucos valores de chave de busca ganharão valores adicionais de forma relativamente rápida.

Em vez de bloquear nós de folha de índice em duas fases, alguns esquemas de controle de concorrência utilizam o **bloqueio do valor de chave** sobre valores de chave individuais, permitindo que outros valores de chave sejam inseridos ou excluídos da mesma folha. O bloqueio do valor de chave, assim, oferece maior concorrência. Porém, usar o bloqueio do valor de chave ingenuamente permitiria a ocorrência do fenômeno fantasma; para impedi-lo, usamos a técnica do **bloqueio da chave seguinte**. Nessa técnica, cada pesquisa de índice precisa bloquear não apenas as chaves encontradas dentro do intervalo (ou a chave única, no caso de uma pesquisa pontual), mas também o valor da próxima chave – ou seja, o valor de chave imediatamente maior que o último valor de chave que estava dentro do intervalo. Além disso, cada inserção precisa bloquear não apenas o valor que é inserido, mas também o valor da chave seguinte. Assim, se a transação tentar inserir um valor que estava dentro do intervalo da pesquisa de índice de outra transação, as duas transações entrariam em conflito sobre o valor de chave seguinte ao valor de chave inserido. De modo semelhante, as exclusões também devem bloquear o valor da chave seguinte para o valor sendo excluído, de modo a garantir que os conflitos com pesquisas de intervalo subsequentes de outras consultas sejam detectados.

Resumo

- Quando várias transações são executadas simultaneamente no banco de dados, a coerência dos dados pode não ser mais preservada. É necessário que o sistema controle a interação entre as transações concorrentes, e esse controle é conseguido por meio de um dentre diversos mecanismos chamados esquemas de *controle de concorrência*.
- Para assegurar a seriação, podemos usar diversos esquemas de controle de concorrência. Todos esses esquemas adiam uma operação ou abortam a transação que emitiu a operação. Os mais comuns são os protocolos de bloqueio, os esquemas de ordenação por timestamp, técnicas de validação e esquemas de múltipla versão.
- Um protocolo de bloqueio é um conjunto de regras que indicam quando uma transação pode bloquear e desbloquear cada um dos itens de dados no banco de dados.
- O protocolo de bloqueio em duas fases permite que uma transação bloqueie um novo item de dados somente se essa transação ainda não tiver desbloqueado qualquer item de dados. O protocolo assegura a seriação, mas não liberdade de impasse. Na ausência de informações referentes à maneira como os itens de dados são acessados, o protocolo de bloqueio em duas fases é necessário e suficiente para garantir a seriação.
- O protocolo de bloqueio estrito em duas fases permite a liberação de bloqueios exclusivos somente no final da transação, a fim de assegurar a facilidade de recuperação e inexistência de cascata dos schedules resultantes. O protocolo de bloqueio rigoroso em duas fases libera todos os bloqueios somente no final da transação.
- Os protocolos de bloqueio baseados em gráfico impõem restrições na ordem em que os itens são acessados, e portanto podem garantir a seriação sem exigir o uso do bloqueio em duas fases, e além disso podem garantir a liberdade de impasse.
- Um esquema de ordenação por timestamp garante a seriação selecionando uma ordenação com antecedência entre cada par de transações. Um timestamp fixo exclusivo é associado a cada transação no sistema. Os timestamps das transações determinam a ordem de seriação. Assim, se o timestamp da transação T_i for menor que o timestamp da transação T_j, então o esquema garante que o schedule produzido é equivalente a um schedule serial em que a transação T_i aparece antes da transação T_j. Ele

faz isso por meio do rollback de uma transação sempre que essa ordem for violada.

- Um esquema de validação é um método de controle de concorrência apropriado nos casos em que uma maioria de transações são transações apenas de leitura, e assim a taxa de conflitos entre essas transações é baixa. Um timestamp fixo exclusivo é associado a cada transação no sistema. A ordem de seriação é determinada pelo timestamp da transação. Uma transação nesse esquema nunca é adiada. Porém, ela precisa passar em um teste de validação para ser concluída. Se ela não passar no teste de validação, o sistema a reverte para o seu estado inicial.
- Existem circunstâncias em que seria vantajoso agrupar diversos itens de dados e tratá-los como um item de dados agregado para fins de trabalho, resultando em vários níveis de granularidade. Permitimos itens de dados de vários tamanhos e definimos uma hierarquia de itens de dados, em que os itens pequenos são aninhados dentro dos maiores. Essa hierarquia pode ser representada graficamente como uma árvore. Os bloqueios são adquiridos na ordem da raiz para a folha; eles são liberados na ordem da folha para a raiz. O protocolo garante a seriação, mas não a liberdade de impasse.
- Um esquema de controle de concorrência de múltipla versão é baseado na criação de uma nova versão de um item de dados para cada transação que escreve esse item. Quando uma operação de leitura é emitida, o sistema seleciona uma das versões para ser lida. O esquema de controle de concorrência garante que a versão a ser lida é selecionada de uma maneira que garanta a seriação, usando timestamps. Uma operação de leitura sempre tem sucesso.
  - Na ordenação de timestamp em múltipla versão, uma operação de escrita pode resultar no rollback da transação.
  - No bloqueio de duas fases em múltipla versão, as operações de escrita podem resultar em uma espera pelo bloqueio ou, possivelmente, em impasse.
- Diversos protocolos de bloqueio não protegem contra impasses. Um modo de evitar o impasse é usar uma ordenação de itens de dados e solicitar bloqueios em uma seqüência consistente com a ordenação.
- Outro modo de impedir impasse é usar preempção e rollbacks de transação. Para controlar a preempção, atribuímos um timestamp exclusivo para cada transação. O sistema usa esses timestamps para decidir se uma transação deverá esperar ou reverter. Se uma transação for revertida, ela reterá seu antigo timestamp quando reiniciada. O esquema ferir-esperar é um esquema preemptivo.
- Se os impasses não forem impedidos, o sistema terá de lidar com eles usando um esquema de detecção e recuperação de impasse. Para fazer isso, o sistema constrói um gráfico de espera. Um sistema está em um estado de impasse se e somente se o gráfico de espera tiver um ciclo. Quando o algoritmo de detecção de impasse determinar que existe um impasse, o sistema terá de se recuperar dele. Ele faz isso revertendo uma ou mais transações para quebrar o impasse.
- Uma operação **delete** só poderá ser realizada se a transação excluindo a tupla tiver um bloqueio exclusivo sobre a tupla a ser excluída. Uma transação que insere uma nova tupla no banco de dados recebe um bloqueio exclusivo sobre a tupla.
- As inserções podem levar ao fenômeno fantasma, em que uma inserção entra em conflito lógico com uma consulta, embora as duas transações possam não acessar uma tupla em comum. Esse conflito não pode ser detectado se o bloqueio for feito apenas sobre tuplas acessadas pelas transações. O bloqueio é exigido sobre os dados usados para encontrar as tuplas na relação. A técnica de bloqueio de índice resolve esse problema exigindo bloqueios sobre certos buckets de índice. Esses bloqueios garante que todas as transações conflitantes entrem em conflito sobre um item de dados real, e não sobre um fantasma.
- Níveis de coerência fracos são usados em algumas aplicações em que a consistência dos resultados da consulta não é crítica, e o uso da seriação resultaria em consultas afetando adversamente o processamento da transação. A consistência de grau dois é uma espécie desse nível de consistência mais fraco; a estabilidade do cursor é um caso especial e muito utilizado de consistência de grau dois. A SQL:1999 permite que as consultas especifiquem o nível de consistência que elas exigem.
- Técnicas especiais de controle de concorrência podem ser desenvolvidas para estruturas de dados especiais. Normalmente, técnicas especiais são aplicadas em árvores B+ para permitir maior concorrência. Essas técnicas permitem o acesso não passível de seriação à árvore B+, mas garantem que a estrutura de árvore B+ esteja correta, assegurando que os acessos ao próprio banco de dados sejam passíveis de seriação.

Termos de revisão

- Controle de concorrência
- Tipos de bloqueio
  - Bloqueio em modo compartilhado (S)
  - Bloqueio em modo exclusivo (X)
- Bloqueio
  - Compatibilidade
  - Solicitação
  - Espera
  - Concessão

- Impasse
- Estagnação
- Protocolo de bloqueio
- Schedule legal
- Protocolo de bloqueio em duas fases
  - Fase de crescimento
  - Fase de encolhimento
  - Ponto de bloqueio
  - Bloqueio estrito em duas fases
  - Bloqueio rigoroso em duas fases
- Conversão de bloqueio
  - Upgrade
  - Downgrade
- Protocolos baseados em gráfico
  - Protocolo de árvore
  - Dependência de commit
- Protocolos baseados em timestamp
- Timestamp
  - Clock do sistema
  - Contador lógico
  - W-timestamp(Q)
  - R-timestamp(Q)
  - Protocolo de ordenação de timestamp
  - Regra do write de Thomas
- Protocolos baseados em validação
  - Fase de leitura
  - Fase de validação
  - Fase de escrita
  - Teste de validação
- Granularidade múltipla
  - Bloqueios explícitos
  - Bloqueios implícitos
  - Bloqueios de intenção
- Modos de bloqueio de intenção
  - Intention-shared (IS)
  - Intention-exclusive (IX)
  - Shared and intention-exclusive (SIX)
- Protocolo de bloqueio por granularidade múltipla
- Controle de concorrência de múltipla versão
- Versões
- Ordenação por timestamp em múltipla versão
- Bloqueio de duas fases em múltipla versão
  - Transações somente de leitura
  - Transações de atualização
- Tratamento de impasse
  - Prevenção
  - Detecção
  - Recuperação
- Prevenção de impasse
  - Bloqueio ordenado
  - Preempção de bloqueios
  - Esquema esperar-morrer
  - Esquema ferir-esperar
  - Esquemas baseados em tempo limite
- Detecção de impasse
  - Gráfico de espera
- Recuperação de impasse
  - Rollback total
  - Rollback parcial
- Operações de inserção e exclusão
- Fenômeno fantasma
  - Protocolo de bloqueio de índice
- Níveis de consistência fracos
  - Consistência de grau dois
  - Estabilidade de cursor
  - Leitura repetitiva
  - Leitura confirmada
  - Leitura não confirmada
- Concorrência nos índices
  - Caranguejo
  - Árvores B-link
  - Protocolo de bloqueio de árvore B-link
  - Bloqueio da chave seguinte

Exercícios práticos

16.1 Mostre que o protocolo de bloqueio em duas fases assegura a seriação de conflito, e que as transações podem ser seriadas de acordo com seus pontos de bloqueio.

16.2 Considere as duas transações a seguir:

T_{31}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{32}: read (B);
read (A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Acrescente instruções de bloqueio e desbloqueio às transações T_{31} e T_{32}, de modo que observem o protocolo de bloqueio em duas fases. A execução dessas transações poderá resultar em um impasse?

16.3 Que benefício o bloqueio rigoroso em duas fases oferece? Como ele se compara com outras formas de bloqueio em duas fases?

16.4 Considere um banco de dados organizado na forma de uma árvore enraizada. Suponha que insiramos um vértice fictício entre cada par de vértices. Mostre que, se seguirmos o protocolo de árvore na nova árvore, receberemos melhor concorrên-

cia do que se seguirmos o protocolo de árvore na árvore original.

16.5 Mostre, por meio de exemplo, que existem schedules possíveis sob o protocolo de árvore que não são possíveis sob o protocolo de bloqueio em duas fases, e vice-versa.

16.6 Considere a seguinte extensão ao protocolo de bloqueio de árvore, que permite bloqueios compartilhados e exclusivos:

- Uma transação pode ser uma transação somente leitura, quando poderá solicitar apenas bloqueios compartilhados, ou uma transação de atualização, quando poderá solicitar apenas bloqueios exclusivos.
- Cada transação precisa seguir as regras do protocolo de árvore. As transações somente leitura podem bloquear qualquer item de dados primeiro, enquanto as transações de atualização tiverem de bloquear a raiz primeiro.

Mostre que o protocolo garante a seriação e a liberdade de impasse.

16.7 Considere o seguinte protocolo de bloqueio baseado em gráfico, que permite apenas modos de bloqueio exclusivos, e que opera sobre os gráficos de dados que estão na forma de um gráfico acíclico direcionado para a raiz.

- Uma transação pode bloquear qualquer vértice primeiro.
- Para bloquear qualquer outro vértice, a transação precisa estar mantendo um bloqueio sobre a maioria dos pais desse vértice.

Mostre que o protocolo garante a seriação e a liberdade de impasse.

16.8 Considere o seguinte protocolo de bloqueio baseado em gráfico, que permite apenas modos de bloqueio exclusivos e que opera sobre gráficos de dados que estão na forma de um gráfico acíclico direcionado para a raiz.

- Uma transação pode bloquear qualquer vértice primeiro.
- Para bloquear qualquer outro vértice, a transação precisa ter visitado todos os pais desse vértice e precisa estar mantendo um bloqueio sobre um dos pais do vértice.

Mostre que o protocolo garante a seriação e a liberdade de impasse.

16.9 O bloqueio não é feito explicitamente nas linguagens de programação persistentes. Em vez disso, os objetos (ou as páginas correspondentes) precisam ser bloqueadas quando os objetos forem acessados. A maioria dos sistemas operacionais modernos permite que o usuário defina proteções de acesso (nenhum acesso, leitura, escrita) sobre páginas, e o acesso à memória que infringe as proteções de acesso resulta em uma violação de proteção (veja o comando mprotect do Unix, por exemplo). Descreva como o mecanismo de proteção de acesso pode ser usado para o bloqueio em nível de página em uma linguagem de programação persistente.

16.10 Considere um sistema de banco de dados que inclui uma operação de **incremento** atômica, além das operações read e write. Considere que V seja o valor do item de dados X. A operação

increment(X) by C

define o valor de X como V + C em uma etapa atômica. O valor de X não está disponível à transação, a menos que essa execute um read(X). A Figura 16.23 mostra uma matriz de compatibilidade de bloqueio para três modos de bloqueio: modo compartilhado, modo exclusivo e modo de incremento.

a. Mostre que, se todas as transações bloquearem os dados que elas acessam no modo correspondente, então o bloqueio em duas fases garante a seriação.

b. Mostre que a inclusão dos bloqueios no modo de **incremento** permite maior concorrência. (Dica: considere as transações de compensação de cheques em nosso exemplo bancário.)

16.11 Na ordenação por timestamp, **W-timestamp**(Q) indica o maior timestamp de qualquer transação que executou `write`(Q) com sucesso. Suponha que, em vez disso, nós a definamos como sendo o timestamp da transação mais recente a executar `write`(Q) com sucesso. Essa mudança de termos faz alguma diferença? Explique sua resposta.

16.12 O uso do bloqueio por granularidade múltipla pode exigir mais ou menos bloqueios do que um sistema equivalente com uma única granularidade de blo-

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

Figura 16.23 Matriz de compatibilidade de bloqueio.

queio. Forneça exemplos das duas situações e compare a quantidade relativa de concorrência permitida.

16.13 Considere o esquema de controle de concorrência baseado em validação da seção "Protocolos baseados em validação". Mostre que, escolhendo Validation(T_i), em vez de Start(T_i), como timestamp da transação T_i, podemos esperar um tempo de resposta melhor, desde que as taxas de conflito entre as transações sejam realmente baixas.

16.14 Para cada um dos protocolos a seguir, descreva aspectos de aplicações práticas que o levariam a sugerir o uso do protocolo, e aspectos que sugeririam não usar o protocolo:

- Bloqueio em duas fases
- Bloqueio em duas fases com bloqueio de múltipla granularidade
- O protocolo de árvore
- Ordenação por timestamp
- Validação
- Ordenação de timestamp em múltipla versão
- Bloqueio em duas fases em múltipla versão

16.15 Explique por que a técnica a seguir para execução de transação pode oferecer melhor desempenho do que apenas usar o bloqueio estrito em duas fases: primeiro, execute a transação sem adquirir quaisquer bloqueios e sem realizar quaisquer escritas no banco de dados, como nas técnicas baseadas em validação, mas, diferente das técnicas de validação, não realize qualquer validação ou escrita no banco de dados. Em vez disso, reexecute a transação usando o bloqueio estrito em duas fases. (Dica: considere as esperas por E/S de disco.)

16.16 Considere o protocolo de ordenação por timestamp e duas transações, uma que escreve dois itens de dados p e q, e outra que lê os mesmos dois itens de dados. Dê um schedule pelo qual o teste de timestamp para uma operação write falha e faz com que a primeira transação seja reiniciada, por sua vez, causando um aborto em cascata da outra transação. Mostre como isso poderia resultar na estagnação das duas transações. (Essa situação, em que dois ou mais processos executam ações, mas são incapazes de completar sua tarefa devido à interação com outros processos, é chamada de **livelock**.)

16.17 Crie um protocolo baseado em timestamp que evita o fenômeno fantasma.

16.18 Suponha que usemos o protocolo de árvore da seção "Protocolos baseados em gráfico" para gerenciar o acesso concorrente a uma árvore B+. Como uma divisão pode ocorrer sobre uma inserção que afeta a raiz, parece que uma operação de inserção não pode liberar quaisquer bloqueios até que tenha completado a operação inteira. Sob que circunstâncias é possível liberar um bloqueio mais cedo?

Exercícios

16.19 Que benefícios o bloqueio estrito em duas fases oferece? Que desvantagens são resultantes?

16.20 A maioria das implementações de sistemas de banco de dados utiliza o bloqueio estrito em duas fases. Sugira três motivos para a popularidade desse protocolo.

16.21 Considere uma variante do protocolo de árvore chamada protocolo de *floresta*. O banco de dados é organizado como uma floresta de árvores enraizadas. Cada transação T_i precisa cumprir as seguintes regras:

- O primeiro bloqueio em cada árvore pode ser sobre qualquer item de dados.
- O segundo bloqueio em uma árvore, e todos os seguintes, só pode ser solicitado se o pai do nó solicitado estiver atualmente bloqueado.
- Os itens de dados podem ser desbloqueados a qualquer momento.
- Um item de dados não pode ser bloqueado novamente por T_i depois de ter sido desbloqueado por T_i.

Mostre que o protocolo de floresta *não* garante a seriação.

16.22 Quando uma transação é revertida sob a ordenação de timestamp, ela recebe um novo timestamp. Por que ela não pode simplesmente manter seu timestamp antigo?

16.23 No bloqueio por granularidade múltipla, qual é a diferença entre bloqueio implícito e explícito?

16.24 Embora o modo SIX seja útil no bloqueio por granularidade múltipla, um modo compartilhado e exclusivo de intenção (XIS) não tem utilidade. Por que isso acontece?

16.25 Mostre que existem schedules que são possíveis sob o protocolo de bloqueio em duas fases, mas não são possíveis sob o protocolo de timestamp, e vice-versa.

16.26 Sob uma versão modificada do protocolo de timestamp, exigimos que um bit de commit seja testado para ver se uma solicitação read precisa esperar. Explique como o bit de commit pode impedir o aborto em cascata. Por que esse teste não é necessário para solicitações write?

16.27 Sob que condições é menos dispendioso evitar o impasse do que permitir que os impasses ocorram e depois detectá-los?

16.28 Se o impasse for evitado por esquemas de impedimento de impasse, a estagnação ainda é possível? Explique sua resposta.

16.29 Explique o fenômeno fantasma. Por que esse fenômeno pode levar a uma execução concorrente incorreta apesar do uso do protocolo de bloqueio em duas fases?

16.30 Explique o motivo para o uso da consistência de grau dois. Que desvantagens essa técnica tem?

16.31 Dê schedules de exemplo para mostrar que, com o bloqueio do valor de chave, se qualquer pesquisa, inserção ou exclusão não bloquear o valor da chave seguinte, o fenômeno fantasma poderia passar sem ser detectado.

16.32 Se muitas transações atualizarem um item comum (por exemplo, o saldo em caixa em uma agência) e itens privados (por exemplo, saldos de conta individuais), explique como você pode aumentar a concorrência (e throughput) ordenando as operações da transação.

16.33 Considere o protocolo de bloqueio a seguir. Todos os itens são numerados, e quando um item estiver desbloqueado, somente itens com número mais alto podem ser bloqueados. Os bloqueios podem ser liberados a qualquer momento. Somente bloqueios X são usados.

Mostre, por meio de um exemplo, que esse protocolo não garante a seriação.

Notas bibliográficas

Gray e Reuter [1993] oferecem um livro-texto detalhado sobre conceitos de processamento de transação, incluindo conceitos de controle de concorrência e detalhes de implementação. Bernstein e Newcomer [1997] oferecem um livro-texto sobre os vários aspectos do processamento de transação, incluindo controle de concorrência.

Os primeiros livros-texto com discussões sobre controle de concorrência e recuperação incluíram Papadimitriou [1986] e Bernstein *et al.* [1987]. Um artigo inicial sobre questões de implementação em controle de concorrência e recuperação é apresentado por Gray [1978].

O protocolo de bloqueio em duas fases foi introduzido por Eswaran *et al.* [1976]. O protocolo de bloqueio em árvore é de Sulberschatz e Kedem [1980]. Outros protocolos de bloqueio não em duas fases que operam sobre gráficos mais gerais são descritos em Yannakakis *et al.* [1979], Kedem e Silberschatz [1983] e Buckley e Silberschatz [1985]. Korth [1983] explora diversos modos de bloqueio que podem ser obtidos pelos modos de bloqueio compartilhado básico e exclusivo.

O Exercício prático 16.4 é de Buckley e Silberschatz [1984]. O Exercício prático 16.6 é de Kedem e Silberschatz [1983]. O Exercício prático 16.7 é de Kedem e Silberschatz [1979]. O Exercício prático 16.8 é de Yannakakis *et al.* [1979]. O Exercício prático 16.10 é de Korth [1983].

O esquema de controle de concorrência baseado em timestamp é de Reed [1983]. Uma exposição de vários algoritmos de controle de concorrência baseados em timestamp é apresentada por Bernstein e Goodman [198]. Um algoritmo de timestamp que não exige qualquer rollback para garantir a seriação é apresentado por Buckley e Silberschatz [1983]. O esquema de controle de concorrência com validação é de Kung e Robinson [1981].

O protocolo de bloqueio para itens de dados com granularidade múltipla é de Gray *et al.* [1975]. Uma descrição detalhada é apresentada por Gray *et al.* [1976]. Korth [1983] formaliza o bloqueio por granularidade múltipla para uma coleção qualquer de modos de bloqueio (permitindo mais semântica do que simplesmente leitura e escrita). Essa técnica inclui uma classe de modos de bloqueio chamada de modos de *atualização*, para lidar com a conversão de bloqueio. Carey [1983] estende a idéia de granularidade múltipla para o controle de concorrência baseado em timestamp. Uma extensão do protocolo para garantir a liberdade de impasse é apresentada por Korth [1982].

Discussões referentes ao controle de concorrência de múltipla versão são oferecidas por Bernstein *et al.* [1983]. Um algoritmo de bloqueio em árvore com múltipla versão aparece em Silberschatz [1982]. A ordenação por timestamp em múltipla versão foi introduzida em Reed [1978] e Reed [1983]. Lai e Wilkinson [1984] descrevem um certificador de bloqueio em duas fases com múltipla versão.

A consistência de grau dois foi introduzida em Gray *et al.* [1975]. Os níveis de consistência – ou isolamento – oferecidos na SQL são explicados e criticados em Berenson *et al.*[1995]. Muitos sistemas de banco de dados comerciais utilizam técnicas baseadas em versão em combinação com o bloqueio. O Oracle usa uma forma de isolamento baseado em snapshot, com base no protocolo descrito na seção "Bloqueio de duas fases em múltipla versão". Os detalhes estão no Capítulo 27 e em Fekete *et al.* [2005]. A técnica de múltipla versão do PostgreSQL é abordada no Capítulo 26, enquanto a técnica de múltipla versão do SQLServer (snapshot) é abordada no Capítulo 29.

A concorrência em árvores B+ foi estudada por Bayer e Schkolnick [1977] e Johnson e Shasha [1993]. As técnicas apresentadas na seção "Concorrência em estruturas de índice" são baseadas em Kung e Lehman [1980] e Lehman e Yao [1981]. A técnica de bloqueio de valor de chave usada no ARIES provê uma concorrência muito alta no acesso por árvore B+ e é descrita em Mohan [1990a] e Levine [1992]. Ellis [1987] apresenta uma técnica de controle de concorrência para o hashing linear.

CAPÍTULO 17

Sistema de recuperação

Um sistema de computador, como qualquer outro dispositivo, está sujeito a falhas por uma série de causas: falha de disco, falta de energia, erro de software, um incêndio na sala e até mesmo sabotagem. Em qualquer falha, informações podem ser perdidas. Portanto, o sistema de banco de dados precisa tomar ações de antemão para garantir que as propriedades de atomicidade e durabilidade das transações, apresentadas no Capítulo 15, sejam preservadas. Uma parte integral de um sistema de banco de dados é um **esquema de recuperação** que pode restaurar o banco de dados ao estado coerente que existia antes da falha. O esquema de recuperação também precisa oferecer **alta disponibilidade**; ou seja, precisa reduzir o tempo durante o qual o banco de dados não pode ser usado após uma falha.

Classificação das falhas

Existem vários tipos de falha que podem ocorrer em um sistema, cada um deles precisa ser tratado de uma maneira diferente. O tipo mais simples de falha é aquela que não resulta na perda de informações no sistema. As falhas que são mais difíceis de tratar são aquelas que resultam em perda de informações. Neste capítulo, vamos considerar apenas os seguintes tipos de falha:

- **Falha de transação**. Existem dois tipos de erros que podem causar a falha de uma transação:
  - **Erro lógico**. A transação não pode mais continuar com sua execução normal devido a alguma condição interna, como entrada defeituosa, dados não encontrados, estouro ou limite de recursos ultrapassado.
  - **Erro do sistema**. O sistema entrou em um estado indesejável (por exemplo, impasse), fazendo com que uma transação não possa continuar com sua execução normal. Contudo, a transação não pode ser reexecutada em outro momento.
- **Falha do sistema**. Existe um defeito do hardware, ou um bug no software de banco de dados ou no sistema operacional, causando a perda do conteúdo do armazenamento volátil e encerrando o processamento da transação. O conteúdo do armazenamento não volátil permanece intacto e não é adulterado.

  A suposição de que erros de hardware e bugs no software fazem o sistema parar, mas não corrompem o conteúdo do armazenamento não volátil, é conhecido como **suposição falhar-parar**. Sistemas bem projetados possuem diversas verificações internas, no nível de hardware e de software, que fazem com que o sistema pare quando existe um erro. Logo, a suposição falhar-parar é razoável.
- **Falha de disco**. Um bloco de disco perde seu conteúdo como resultado de uma falha da cabeça ou uma falha durante uma operação de transferência de dados. As cópias dos dados em outros discos, ou backups de arquivamento em mídia terciária, como fitas, são usadas para a recuperação de falhas.

Para determinar como o sistema deverá se recuperar de falhas, precisamos identificar os modos de falha daqueles dispositivos usados para armazenamento de dados. Em seguida, temos de considerar como esses modos de falha afetam o conteúdo do banco de dados. Podemos, então, propor algoritmos para garantir a coerência do banco de dados e a atomicidade da transação apesar das falhas. Esses algoritmos, conhecidos como algoritmos de recuperação, possuem duas partes:

1. Ações tomadas durante o processamento de transação normal, para garantir que haja informações suficientes para permitir a recuperação de falhas.
2. Ações tomadas após uma falha, para recuperar o conteúdo do banco de dados a um estado que garanta a consistência do banco de dados, atomicidade da transação e durabilidade.

Estrutura de armazenamento

Como vimos no Capítulo 11, os diversos itens de dados no banco de dados podem ser armazenados em diversos meios de armazenamento diferentes. Para entender como garantir as propriedades de atomicidade e durabilidade de uma transação, temos de compreender esses meios de armazenamento e seus métodos de acesso.

Tipos de armazenamento

No Capítulo 11, vimos que o meio de armazenamento pode ser distinguido por sua velocidade relativa, capacidade e poder de recuperação de falhas, e pode ser classificado como armazenamento volátil ou armazenamento não volátil. Revisamos esses termos e apresentamos outra classe de armazenamento, chamada **armazenamento estável**.

- **Armazenamento volátil**. As informações residindo no armazenamento volátil normalmente não sobrevivem a falhas do sistema. Alguns exemplos desse tipo de armazenamento são a memória principal e a memória cache. O acesso ao armazenamento volátil é extremamente rápido, tanto por causa da velocidade do próprio acesso à memória, quanto porque é possível acessar qualquer item de dados no armazenamento volátil diretamente.
- **Armazenamento não volátil**. As informações residindo no armazenamento não volátil sobrevivem a falhas do sistema. Alguns exemplos desse tipo de armazenamento são discos e fitas magnéticas. Os discos são usados para armazenamento on-line, enquanto as fitas são usadas para armazenamento de arquivamento. Porém, ambos estão sujeitos a falhas (por exemplo, falha de cabeça), que podem resultar em perda de informações. No estado atual da tecnologia, o armazenamento não volátil é mais lento que o armazenamento volátil por várias ordens de grandeza. Isso porque os dispositivos de disco e fita são eletromecânicos, em vez de serem baseados inteiramente em chips, como no armazenamento volátil. Nos sistemas de banco de dados, os discos são usados para a maioria do armazenamento não volátil. Outros meios não voláteis normalmente são usados apenas para dados de backup. O armazenamento flash (ver seção "Visão geral do meio de armazenamento físico" do Capítulo 11), embora não volátil, tem capacidade suficiente para a maioria dos sistemas de banco de dados.
- **Armazenamento estável**. Informações residindo no armazenamento estável *nunca* são perdidas (*nunca* deve ser algo assumido com uma certa desconfiança, pois teoricamente não pode ser garantido – por exemplo, é possível, embora extremamente improvável, que um buraco negro possa engolir a Terra e destruir permanentemente todos os dados!). Embora o armazenamento estável seja teoricamente impossível de se obter, ele pode ser aproximado por técnicas que tornam a perda de dados extremamente improvável. A próxima seção discute a implementação do armazenamento estável.

As distinções entre os diversos tipos de armazenamento normalmente são menos claras na prática do que em nossa apresentação. Certos sistemas oferecem backup com bateria, de modo que alguma memória principal poderá sobreviver a falhas do sistema e faltas de energia. Formas alternativas de armazenamento não volátil, como a mídia óptica, oferecem um grau de confiabilidade ainda mais alto do que os discos.

Implementação do armazenamento estável

Para implementar o armazenamento estável, precisamos replicar a informação necessária em vários meios de armazenamento não voláteis (normalmente, disco) com modos de falha independentes e atualizar a informação de uma maneira controlada para garantir que a falha durante a transferência de dados não danifique a informação necessária.

Lembre-se (pelo Capítulo 11) que os sistemas RAID garantem que a falha de um único disco (mesmo durante a transferência de dados) não resultará em perda de dados. A forma mais rápida e mais simples de RAID é o disco espelhado, que mantém duas cópias de cada bloco em discos separados. Outras formas de RAID oferecem custos mais baixos, mas ao custo de um desempenho inferior.

Sistemas RAID, porém, não podem proteger contra perda de dados devido a desastres como incêndios ou inundação. Muitos sistemas armazenam backups de arquivamento de fitas fora da instalação, para proteção contra tais desastres. Porém, como as fitas não podem ser transportadas para fora da instalação continuamente, as atualizações desde o momento mais recente em que as fitas foram retiradas de lá poderiam ser perdidas em tal desastre. Sistemas mais seguros mantêm uma cópia de cada bloco de armazenamento estável em uma instalação remota, gravando-a por fora por meio de uma rede de computadores, além de armazenar o bloco em um sistema de disco local. Como os blo-

cos são gerados para um sistema remoto quando são gerados para o armazenamento local, uma vez que a operação de saída seja concluída, a saída não se perde, mesmo no caso de um desastre como um incêndio ou uma inundação. Estudamos tais sistemas de *backup remoto* na seção "Sistemas de backup remoto".

No restante desta seção, discutimos como o meio de armazenamento pode ser protegido contra falha durante a transferência de dados. A transferência em bloco entre a memória e o armazenamento em disco pode resultar em

- **Término bem-sucedido**. A informação transferida chegou com segurança em seu destino.
- **Falha parcial**. A falha ocorreu no meio da transferência, e o bloco de destino possui informações incorretas.
- **Falha total**. A falha ocorreu suficientemente cedo durante a transferência, de modo que o bloco de destino permanece intacto.

É preciso que, se houver uma **falha de transferência de dados**, o sistema a detecte e invoque um procedimento de recuperação para restaurar o bloco a um estado consistente. Para isso, o sistema precisa manter dois blocos físicos para cada bloco lógico do banco de dados; no caso de discos espelhados, os dois blocos estão no mesmo local; no caso de backup remoto, um dos blocos é local, enquanto o outro está em uma instalação remota. Uma operação de saída é executada da seguinte maneira:

1. Escreva a informação no primeiro bloco físico.
2. Quando a primeira escrita for terminada com sucesso, escreva a informação no segundo bloco físico.
3. A saída só é completada depois que a segunda escrita terminar com sucesso.

Durante a recuperação, o sistema examina cada par de blocos físicos. Se ambos forem iguais e não houver um erro detectável, então nenhuma outra ação é necessária. (Lembre-se de que os erros em um bloco de disco, como uma escrita parcial no bloco, são detectados armazenando uma soma de verificação com cada bloco.) Se o sistema detectar um erro em um bloco, então ele substitui seu conteúdo pelo conteúdo do outro bloco. Se os dois blocos não tiverem um erro detectável, mas diferirem em conteúdo, então o sistema substitui o conteúdo do primeiro bloco pelo valor do segundo. Esse procedimento de recuperação garante que uma escrita no armazenamento estável tenha sucesso completo (ou seja, atualize todas as cópias) ou resulte em nenhuma mudança.

O requisito de comparar cada par de blocos correspondente durante a recuperação é dispendioso para ser atendido. Podemos reduzir bastante o custo registrando as escritas de bloco que estão em andamento, usando uma pequena quantidade de memória RAM não volátil. Na recuperação, somente os blocos para os quais as escritas estavam em progresso precisam ser comparados.

Os protocolos para escrever um bloco em uma instalação remota são semelhantes aos protocolos para escrever blocos em um sistema de disco espelhado, que examinamos no Capítulo 11 e particularmente no Exercício prático 11.2.

É fácil estender esse procedimento para permitir o uso de um número arbitrariamente grande de cópias de cada bloco de armazenamento estável. Embora uma grande quantidade de cópias reduza a probabilidade de uma falha para menos ainda do que com duas cópias, normalmente é razoável simular o armazenamento estável com apenas duas cópias.

Acesso aos dados

Como vimos no Capítulo 11, o sistema de banco de dados reside permanentemente no armazenamento não volátil (normalmente, discos) e é particionado em unidades de armazenamento de tamanho fixo, chamadas **blocos**. Os blo-

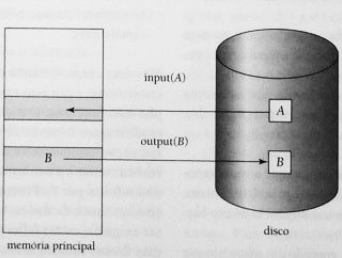


Figura 17.1 Operações de armazenamento em bloco.

cos são as unidades de transferência de dados de e para o disco, e podem conter vários itens de dados. Vamos considerar que nenhum item de dados se espalha por dois ou mais blocos. Essa suposição é realista para a maioria das aplicações de processamento de dados, como nosso exemplo bancário.

As transações incluem informações do disco na memória principal e depois enviam a informação para o disco. As operações de entrada e saída são feitas em unidades de bloco. Os blocos residindo no disco são chamados de **blocos físicos**; os blocos residindo temporariamente na memória principal são conhecidos como **blocos de buffer**. A área da memória em que os blocos residem temporariamente é chamada de **buffer de disco**.

Os movimentos de bloco entre o disco e a memória principal são iniciados por meio das duas operações a seguir:

1. `input`(B) transfere o bloco físico B para a memória principal.
2. `output`(B) transfere o bloco de buffer B para o disco e substitui o bloco físico apropriado.

Cada transação T_i possui uma área de trabalho privada em que são mantidas as cópias de todos os itens de dados acessados e atualizados por T_i. O sistema cria essa área de trabalho quando a transação é iniciada; o sistema a remove quando a transação é confirmada ou abortada. Cada item de dados X mantido na área de trabalho da transação T_i é indicado por x_i. A transação T_i interage com o sistema de banco de dados transferindo dados de e para sua área de trabalho no buffer do sistema. Transferimos dados por meio destas duas operações:

1. `read`(X) atribui o valor do item de dados X à variável local x_i. Ela executa essa operação da seguinte maneira:
   a. Se o bloco B_X em que X reside não estiver na memória principal, emite input(B_X).
   b. Atribui a x_i o valor de X vindo do bloco de buffer.
2. `write`(X) atribui o valor da variável local x_i ao item de dados X no bloco de buffer. Ela executa essa operação da seguinte maneira:
   a. Se o bloco B_X em que X reside não estiver na memória principal, ela emite `input`(B_X).
   b. Atribui o valor de x_i a X no buffer B_X.

Observe que as duas operações podem exigir a transferência de um bloco do disco para a memória principal. Porém, elas não exigem especificamente a transferência de um bloco da memória principal para o disco.

Um bloco de buffer, por fim, é gravado em disco porque o gerenciador de buffer precisa do espaço da memória para outras finalidades ou porque o sistema de banco de dados deseja refletir a mudança feita em B no disco. Diremos que o sistema de banco de dados realiza uma **saída forçada** do buffer B se emitir um `output`(B).

Quando uma transação precisa acessar um item de dados X pela primeira vez, deve executar `read`(X). O sistema, então, realiza todas as atualizações em X sobre x_i. Depois que a transação acessar X pela última vez, ela precisa executar `write`(X) para refletir a mudança feita em X no próprio banco de dados.

A operação `output`(B_X) para o bloco de buffer B_X em que X reside não precisa ter efeito imediatamente depois que `write`(X) for executado, pois o bloco B_X pode conter outros itens de dados que ainda estão sendo acessados. Assim, a saída real pode ocorrer mais tarde. Observe que, se o sistema falhar depois que a operação `write`(X) foi executada, mas antes que `output`(B_X) fosse executada, o novo valor de X nunca é gravado em disco e, portanto, é perdido.

Recuperação e atomicidade

Considere novamente o nosso sistema bancário simplificado e a transação T_i que transfere \$50 da conta A para a conta B, com os valores iniciais de A e B sendo \$1.000 e \$2.000, respectivamente. Suponha que uma falha do sistema tenha ocorrido durante a execução de `output`(T_i), após a execução de `output`(B_A) mas antes da execução de `output`(B_B), onde B_B e B_A indicam os blocos de buffer em que A e B residem. Como o conteúdo da memória foi perdido, não conhecemos o destino da transação; assim, poderíamos invocar um dentre dois procedimentos de recuperação possíveis:

- Executar T_i **novamente**. Esse procedimento resultará no valor de A se tornando \$900, em vez de \$950. Assim, o sistema entra em um estado inconsistente.
- **Não executar** T_i **novamente**. O estado atual do sistema tem os valores de \$950 e \$2.000 para A e B, respectivamente. Assim, o sistema entra em um estado inconsistente.

Nos dois casos, o banco de dados fica em um estado inconsistente, e por isso esse esquema de recuperação simples não funciona. O motivo para essa dificuldade é que modificamos o banco de dados sem ter certeza de que a transação realmente será confirmada. Nosso objetivo é realizar todas ou nenhuma das modificações ao banco de dados feitas por T_i. Porém, se T_i realizou várias modificações no banco de dados, várias operações de saída podem ser exigidas, e uma falha poderá ocorrer depois que algumas dessas modificações tiverem sido feitas, mas antes que todas sejam feitas.

Para alcançar nosso objetivo de atomicidade, primeiro temos de emitir informações descrevendo as modificações feitas no armazenamento estável, sem modificar o próprio banco de dados. Como veremos, esse procedimento nos permitirá gerar todas as modificações feitas por uma transação confirmada, apesar das falhas. Vamos considerar que as transações sejam executadas de forma serial; em outras palavras, somente uma única transação está ativa ao mesmo tempo. Vamos descrever como tratar da execução simultânea das transações mais adiante, na seção "Recuperação com transações concorrentes".

Recuperação baseada em log

A estrutura mais utilizada para registrar as modificações do banco de dados é o log. O log é uma seqüência de registros de log, registrando todas as atividades de atualização no banco de dados. Existem vários tipos de registros de log. Um registro de log de atualização descreve uma única escrita no banco de dados. Ele tem estes campos:

- O **identificador de transação** é o identificador exclusivo da transação que realizou a operação write.
- O **identificador de item de dados** é o identificador exclusivo do item de dados escrito. Normalmente, esse é o local no disco do item de dados.
- O **valor antigo** é o valor do item de dados antes da escrita.
- O **novo valor** é o valor que o item de dados terá após a escrita.

Outros registros de log especiais existem para registrar eventos significativos durante o processamento da transação, como o início de uma transação e a confirmação ou aborto de uma transação. Indicamos os vários tipos de registros de log como:

- $<T_i$ `start`$>$. A transação T_i foi iniciada.
- $<T_i, X_j, V_1, V_2>$. A transação T_i realizou uma escrita sobre o item de dados X_j. X_j tinha valores V_1 antes da escrita e terá o valor V_2 após a escrita.
- $<T_i$ `commit`$>$. A transação T_i foi confirmada.
- $<T_i$ `abort`$>$. A transação T_i foi abortada.

Sempre que uma transação realiza uma escrita, é essencial que o registro de log para essa escrita seja criado antes que o banco de dados seja modificado. Quando existe um registro de log, podemos emitir a modificação no banco de dados se isso for desejado. Além disso, temos a capacidade de desfazer uma modificação que já foi enviada ao banco de dados. Isso é feito usando o campo do valor antigo nos registros de log.

Para os registros de log serem úteis à recuperação do sistema e a falhas de disco, o log precisa residir no armazenamento estável. Por enquanto, vamos considerar que cada registro de log é escrito no final do log, no armazenamento estável, assim que é criado. Na seção "Gerenciamento de buffer", veremos quando é seguro relaxar essa exigência para reduzir a sobrecarga imposta pelo logging. Nas seções "Modificação de banco de dados adiada" e "Modificação imediata do banco de dados", apresentaremos duas técnicas para usar o log a fim de garantir a atomicidade da transação apesar das falhas. Observe que o log contém um registro completo de toda a atividade do banco de dados. Como resultado, o valor de dados armazenados no log pode se tornar incrivelmente grande. Na seção "Pontos de verificação (Check points)", mostraremos quando é seguro apagar a informação do log.

Modificação de banco de dados adiada

A **técnica de modificação adiada** garante a atomicidade da transação, registrando todas as modificações do banco de dados no log, mas adiando a execução de todas as operações write de uma transação até que a transação seja parcialmente confirmada. Lembre-se de que uma transação é considerada parcialmente confirmada quando a ação final da transação tiver sido executada. A versão da técnica de modificação adiada que descrevemos nesta seção considera que as transações são executadas em série.

Quando uma transação está parcialmente confirmada, a informação no log associado à transação é usada na execução das escritas adiadas. Se o sistema falhar antes que a transação termine sua execução, ou se a transação abortar, então a informação no log é simplesmente ignorada.

A execução da transação T_i prossegue da seguinte maneira. Antes que T_i inicie sua execução, um registro $<T_i$ `start`$>$ é escrito no log. Uma operação `write`(X) por T_i resulta na escrita de um novo registro no log. Finalmente, quando T_i é confirmada parcialmente, um registro $<T_i$ `commit`$>$ é escrito no log.

Quando a transação T_i é confirmada parcialmente, os registros associados a ela no log são usados na execução das escritas desejadas. Como uma falha pode ocorrer enquanto essa atualização está ocorrendo, temos de garantir que, antes do início dessas atualizações, todos os registros de log sejam escritos no armazenamento estável. Quando tiverem sido escritos, a atualização real ocorre, e a transação entra no estado confirmado.

Observe que somente o novo valor do item de dados é exigido pela técnica de modificação adiada. Assim, podemos simplificar a estrutura de registro de log de atualização que vimos na seção anterior, omitindo o campo de valor antigo. Para ilustrar, reconsidere nosso sistema bancário simplificado. Considere que T_0 seja uma transação que transfere 50 da conta A para a conta B:

<T_0 start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0, commit>
<T_1 start>
<T_1, C, 600>
<T_1, commit>

Figura 17.2 Parte do log de banco de dados correspondente a T_0 e T_1.

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Seja T_1 uma transação que retira $100 da conta C:

T_1: read(C);
$C := C - 100$;
write(C).

Suponha que essas transações são executadas em série, na ordem T_0 seguida por T_1, e que os valores das contas A, B e C antes que a execução ocorra fossem $1.000, $2.000 e $700, respectivamente. A parte do log contendo as informações relevantes nessas duas transações aparece na Figura 17.2.

Existem várias ordens em que as saídas reais podem ocorrer no sistema de banco de dados e no log como resultado da execução de T_0 e T_1. Uma delas aparece na Figura 17.3. Observe que o valor de A é mudado no banco de dados somente depois que o registro <T_0, A, 950> foi colocado no log.

Usando o log, o sistema pode lidar com qualquer falha que resulte na perda de informações no armazenamento volátil. O esquema de recuperação utiliza o seguinte procedimento de recuperação:

- `redo`(T_i) define o valor de todos os itens de dados atualizados pela transação T_i para os novos valores.

O conjunto de itens de dados atualizados por T_i e seus respectivos novos valores podem ser encontrados no log.

A operação redo precisa ser **idempotente**; ou seja, executá-la várias vezes precisa ser equivalente à execução uma vez. Essa característica é exigida se tivermos que garantir o comportamento correto mesmo que ocorra uma falha durante o processo de recuperação.

Após uma falha, o subsistema de recuperação consulta o log para determinar quais transações precisam ser refeitas. A transação T_i precisa ser refeita se e somente se o log contém o registro <T_i `start`> e o registro <T_i `commit`>. Assim, se o sistema falhar após a transação terminar sua execução, o esquema de recuperação usa a informação no log para restaurar o sistema a um estado consistente anterior após a transação ter completado.

Log	Banco de dados
<T_0 start>	
<T_0, A, 950>	
<T_0 B, 2050>	
<T_0, commit>	
	A = 950
	B = 2050
<T_1start>	
<T_1, C, 600>	
<T_1, commit>	
	C = 600

Figura 17.3 Estado do log e banco de dados correspondente a T_0 e T_1.

Como ilustração, vamos retornar ao nosso exemplo bancário com as transações T_0 e T_1 executadas uma após a outra na ordem T_0 seguida por T_1. A Figura 17.2 mostra o log que resulta da execução completa de T_0 e T_1. Vamos supor que o sistema falhe antes do término das transações, de modo que possamos ver como a técnica de recuperação restaura o banco de dados a um estado consistente. Suponha que a falha ocorra logo após o registro de log para a etapa

```
write(B)
```

da transação T_0 ter sido escrita em armazenamento estável. O log no momento da falha aparece na Figura 17.4a. Quando o sistema retornar, nenhuma ação redo precisa ser tomada, pois nenhum registro commit aparece no log. Os valores das contas A e B continuam sendo \$1.000 e \$2.000, respectivamente. Os registros de log da transação incompleta T_1 podem ser excluídos do log.

Agora, vamos supor que a falha venha imediatamente antes que o registro de log para a etapa

```
write(C)
```

da transação T_1 tenha sido escrito no armazenamento estável. Nesse caso, o log no momento da falha é como na Figura 17.4b. Quando o sistema retornar, a operação redo(T_0) será realizada, pois o registro

```
<T0 commit>
```

aparece no log do disco. Depois que essa operação for executada, os valores das contas A e B são \$950 e \$2.050, respectivamente. O valor da conta C permanece \$700. Como antes, os registros de log da transação incompleta T_1 podem ser excluídos do log.

Finalmente, considere que uma falha ocorre logo após o registro de log

```
<T1 commit>
```

ser escrito no armazenamento estável. O log no momento dessa falha está na Figura 17.4c. Quando o sistema retornar, dois registros commit estão no log: um para T_0 e um para T_1. Portanto, o sistema precisa realizar operações redo(T_0) e redo(T_1), na ordem em que seus registros commit aparecem no log. Após o sistema executar essas operações, os valores das contas A, B e C são \$950, \$2.050 e \$600, respectivamente. Finalmente, vamos considerar um caso em que uma segunda falha do sistema ocorre durante a recuperação da falha do disco. Algumas mudanças podem ter sido feitas no banco de dados como resultado das operações redo, mas nem todas as mudanças podem ter sido feitas. Quando o sistema retornar após a segunda falha, a recuperação prossegue exatamente como nos exemplos anteriores. Para cada registro commit

```
<Ti commit>
```

encontrado no log, o sistema realiza a operação redo(T_i). Em outras palavras, ele reinicia as ações de recuperação desde o início. Como redo escreve valores no banco de dados independente dos valores atualmente no banco de dados, o resultado de uma segunda tentativa bem-sucedida no `redo` é igual ao que seria se `redo` tivesse tido sucesso na primeira vez.

Modificação imediata do banco de dados

As **técnicas de modificação imediata** permitem que as modificações no banco de dados sejam enviadas ao banco de dados enquanto a transação ainda está no estado ativo. As modificações de dados escritas pelas transações ativas são chamadas **modificações não confirmadas**. No evento de uma falha ou falha da transação, o sistema precisa usar o campo do valor antigo dos registros de log, descrito na seção "Recuperação baseada em log", para restaurar os itens de dados modificados para o valor que tinham antes do início da transação. A operação undo, descrita a seguir, realiza essa restauração.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0, commit>	<T_0, commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1, commit>

Figura 17.4 O mesmo log da Figura 17.3, mostrado em três momentos diferentes.

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0, commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1, commit>

Figura 17.5 Parte do log do sistema correspondente a T_0 e T_1.

Antes que uma transação T_i inicie sua execução, o sistema escreve o registro <T_i `start`> no log. Durante sua execução, qualquer operação `write`(X) por T_i é *precedida* pela escrita do novo registro de atualização apropriado no log. Quando T_i for parcialmente confirmada, o sistema escreverá o registro <T_i `commit`> no log.

Como a informação no log é usada na reconstrução do estado do banco de dados, não podemos permitir que a atualização real no banco de dados ocorra antes que o registro de log correspondente seja escrito no armazenamento estável. Portanto, exigimos que, antes da execução de uma operação `output`(B), os registros de log correspondentes a B sejam escritos no armazenamento estável. Retornaremos a essa questão na seção "Gerenciamento de buffer".

Como ilustração, vamos reconsiderar nosso sistema bancário simplificado, com as transações T_0 e T_1 executadas uma após a outra na ordem T_0 seguida por T_1. A parte do log contendo as informações relevantes referentes a essas duas transações aparece na Figura 17.5.

A Figura 17.6 mostra uma ordem possível em que as saídas reais ocorreram no sistema de banco de dados e no log, como resultado da execução de T_0 e T_1. Observe que essa ordem não pôde ser obtida na técnica de modificação adiada da seção anterior.

Usando o log, o sistema pode lidar com qualquer falha que não resulte na perda de informações no armazenamento não volátil. O esquema de recuperação utiliza dois procedimentos de recuperação:

- `undo`(T_i) restaura o valor de todos os itens de dados atualizados pela transação T_i para os valores antigos.
- `redo`(T_i) define o valor de todos os itens de dados atualizados pela transação T_i como os novos valores.

O conjunto de itens de dados atualizados por T_i e seus respectivos valores antigo e novo podem ser encontrados no log.

As operações `undo` e `redo` precisam ser idempotentes para garantir o comportamento correto mesmo que ocorra uma falha durante o processo de recuperação.

Depois que houve uma falha, o esquema de recuperação consulta o log para determinar quais transações precisam ser refeitas e quais precisam ser desfeitas:

- A transação T_i precisa ser desfeita se o log contém o registro <T_i `start`>, mas não contém o registro <T_i `commit`>.
- A transação T_i precisa ser refeita se o log tiver o registro <T_i `start`> e o registro <T_i `commit`>.

Como ilustração, retorne ao nosso exemplo de banco, com as transações T_0 e T_1 executadas uma após a outra na ordem T_0 seguida por T_1. Suponha que o sistema falhe antes do término das transações. Vamos considerar três casos.

Log	Banco de dados
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	A = 950
	B = 2050
<T_0, commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	C = 600
<T_1, commit>	

Figura 17.6 Estado do log do sistema e banco de dados correspondente a T_0 e T_1.

O estado dos logs para cada um desses casos aparece na Figura 17.7.

Primeiro, vamos supor que a falha ocorra logo após o registro de log para a etapa

`write`(B)

da transação T_0 tenha sido escrito no armazenamento estável (Figura 17.7a). Quando o sistema volta à atividade, ele encontra o registro <T_0 start> no log, mas nenhum registro <T_0 commit> correspondente. Assim, a transação T_0 precisa ser desfeita, de modo que um undo(T_0) é realizado. Como resultado, os valores nas contas A e B (no disco) são restaurados para $1.000 e $2.000, respectivamente.

Em seguida, vamos supor que a falha venha imediatamente após o registro de log para a etapa

`write`(C)

da transação T_1 ter sido escrito no armazenamento estável (Figura 17.7b). Quando o sistema volta à atividade, duas ações de recuperação precisam ser tomadas. A operação undo(T_1) precisa ser realizada, pois o registro <T_1 `start`> aparece no log, mas não existe o registro <T_1 `commit`>. A operação `redo`(T_0) precisa ser realizada, pois o log contém o registro <T_0 `start`> e o registro <T_0 `commit`>. Ao final do procedimento de recuperação inteiro, os valores das contas A, B e C são $950, $2.050 e $700, respectivamente. Observe que a operação undo(T_1) é realizada antes do `redo`(T_0). Nesse exemplo, haveria o mesmo resultado se a ordem fosse invertida. No entanto, a ordem de realização de operações undo primeiro e depois as operações redo é importante para o algoritmo de recuperação que veremos na seção "Recuperação com transações concorrentes".

Finalmente, vamos supor que a falha ocorra imediatamente após o registro de log

<T_1 `commit`>

ter sido escrito no armazenamento estável (Figura 17.7c). Quando o sistema volta à atividade, tanto T_0 quanto T_1 precisam ser refeitos, pois os registros <T_0 `start`> e <T_0 `commit`> aparecem no log, assim como os registros <T_1 `start`> e <T_1 `commit`>. Depois que o sistema realiza os procedimentos de recuperação `redo`(T_0) e `redo`(T_1), os valores nas contas A, B e C são $950, $2.050 e $600, respectivamente.

Pontos de verificação (Check points)

Quando ocorre uma falha do sistema, é preciso consultar o log para determinar as transações que precisam ser refeitas e as que precisam ser desfeitas. Em princípio, precisamos pesquisar o log inteiro para determinar essa informação. Existem duas dificuldades importantes com essa técnica:

1. O processo de busca é demorado.
2. A maioria das transações que, de acordo com nosso algoritmo, precisam ser refeitas já enviaram suas atualizações para o banco de dados. Embora refazê-las não cause dano algum, a recuperação levará mais tempo.

Para reduzir esses tipos de sobrecarga, introduzimos os pontos de verificação (*checkpoints*). Durante a execução, o sistema mantém o log, usando uma das duas técnicas descritas nas seções "Modificação de banco de dados adiada" e "Modificação imediata do banco de dados". Além disso, o sistema periodicamente realiza **pontos de verificação**, que exigem a seguinte seqüência de ações:

1. Enviar para o armazenamento estável todos os registros atualmente residindo na memória principal.
2. Enviar para o disco todos os blocos de buffer modificados.
3. Enviar para o armazenamento estável um registro de log <`checkpoint`>.

As transações não têm permissão para realizar quaisquer ações de atualização, como escrever em um bloco de buffer ou escrever um registro de log, enquanto um ponto de verificação está em andamento.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0, commit>	<T_0, commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1, commit>

Figura 17.7 O mesmo log, mostrado em três tempos diferentes.

A presença de um registro <checkpoint> no log permite que o sistema agilize seu procedimento de recuperação. Considere uma transação T_i que foi confirmada antes do ponto de verificação. Para tal transação, o registro <T_i commit> aparece no log antes do registro <checkpoint>. Quaisquer modificações no banco de dados feitas por T_i precisam ter sido escritas no banco de dados antes do ponto de verificação ou como parte do próprio ponto de verificação. Assim, no momento da recuperação, não é preciso realizar uma operação redo sobre T_i.

Essa observação nos permite refinar nossos esquemas de recuperação anteriores. (Continuamos a considerar que as transações são executadas em série.) Depois que houve uma falha, o esquema de recuperação examina o log para determinar a transação T_i mais recente que iniciou a execução antes que ocorresse o ponto de verificação mais recente. Ele poderá encontrar essa transação pesquisando o log ao contrário, a partir do fim do log, até encontrar o primeiro registro <checkpoint> (como estamos pesquisando para trás, o registro encontrado é o registro <checkpoint> final no log); depois, ele continua a pesquisa para trás, até encontrar o próximo registro <T_i start>. Esse registro identifica uma transação T_i.

Quando o sistema tiver identificado a transação T_i, as operações redo e undo precisam ser aplicadas somente à transação T_i e todas as transações T_j que iniciaram a execução após a transação T_i. Vamos indicar essas transações pelo conjunto T. O restante (parte anterior) do log pode ser ignorado e pode ser apagado sempre que se desejar. As operações de recuperação exatas a serem realizadas dependem da técnica de modificação sendo usada. Para a técnica de modificação imediata, as operações de recuperação são:

- Para todas as transações T_k em T que não possuem registro <T_k commit> no log, execute undo(T_k).
- Para todas as transações T_k em T tais que o registro <T_k commit> aparece no log, execute redo(T_k).

Obviamente, a operação undo não precisa ser aplicada quando a técnica de modificação adiada está sendo empregada.

Como ilustração, considere o conjunto de transações $\{T_0, T_1, \ldots, T_{100}\}$ executado na ordem dos subscritos. Suponha que o ponto de verificação mais recente tenha ocorrido durante a execução da transação T_{67}. Assim, somente as transações $T_{67}, T_{68}, \ldots, T_{100}$ precisam ser consideradas durante o esquema de recuperação. Cada uma delas precisa ser refeita se tiver sido confirmada; caso contrário, ela precisa ser desfeita.

Na seção "Pontos de verificação", consideramos uma extensão da técnica de ponto de verificação para o processamento simultâneo da transação.

Recuperação com transações concorrentes

Até agora, consideramos a recuperação em um ambiente em que somente uma única transação de cada vez está executando. Agora, discutimos como podemos modificar e estender o esquema de recuperação baseado em log para lidar com várias transações simultâneas. Independente do número de transações simultâneas, o sistema tem um único buffer de disco e um único log. Todas as transações compartilham os blocos de buffer. Permitimos a modificação imediata e permitimos que um bloco do buffer tenha itens de dados atualizados por uma ou mais transações.

Interação com controle de concorrência

O esquema de recuperação depende muito do esquema de controle de concorrência que é usado. Para reverter uma transação que falhou, temos de desfazer as atualizações realizadas pela transação. Suponha que uma transação T_0 tenha de ser revertida e um item de dados Q que foi atualizado por T_0 tenha de ser restaurado ao seu valor antigo. Usando os esquemas baseados em log para a recuperação, restauramos o valor usando a informação de undo em um registro e log. Suponha agora que uma segunda transação T_1 tenha realizado outra atualização em Q *antes* que T_0 seja revertida. Então, a atualização realizada por T_1 será perdida se T_0 for revertida.

Portanto, é preciso que, se uma transação T tiver atualizado um item de dados Q, nenhuma outra transação possa atualizar o mesmo item de dados até que T tenha sido confirmada ou revertida. Podemos assegurar esse requisito com facilidade usando o bloqueio estrito em duas fases – ou seja, o bloqueio em duas fases com bloqueios exclusivos até o final da transação.

Rollback da transação

Revertemos uma transação que falhou, T_i, usando o log. O sistema varre o log ao contrário; para cada registro de log da forma <T_i, X_j, V_1, V_2> encontrado no log, o sistema restaura o item de dados X_j ao seu valor V_1. A varredura do log termina quando o registro de log <T_i, start> for encontrado.

A varredura do log ao contrário é importante, pois uma transação pode ter atualizado um item de dados mais de uma vez. Como ilustração, considere o par de registros de log

<T_i, A, 10, 20>
<T_i, A, 20, 30>

Os registros de log representam uma modificação do item de dados A por T_i, seguida por outra modificação de A por T_i. A varredura do log ao contrário define A corretamente

para 10. Se o log fosse varrido na direção normal, A seria definido como 20, o que é incorreto.

Se o bloqueio estrito em duas fases for usado para controle de concorrência, os bloqueios mantidos por uma transação T só podem ser liberados após a transação ter sido revertida, conforme descrevemos. Quando a transação T (que está sendo revertida) tiver atualizado um item de dados, nenhuma outra transação poderia ter atualizado o mesmo item de dados, devido aos requisitos de controle de concorrência mencionados na seção anterior. Portanto, restaurar o valor antigo do item de dados não apagará os efeitos de qualquer outra transação.

Pontos de verificação

Na seção "Pontos de verificação (Check points)", usamos pontos de verificação para reduzir o número de registros de log que o sistema precisa varrer quando se recupera de uma falha. Como assumimos que não havia concorrência, foi preciso considerar apenas as seguintes transações durante a recuperação:

- Aquelas que iniciaram após o ponto de verificação mais recente
- A única transação, se houver, que estava ativa no momento do ponto de verificação mais recente

A situação é mais complexa quando as transações podem ser executadas simultaneamente, pois várias transações podem ter estado ativas no momento do ponto de verificação mais recente.

Em um sistema de processamento de transação concorrente, exigimos que o registro de log do ponto de verificação seja da forma <checkpoint L>, onde L é uma lista de transações ativas no momento do ponto de verificação. Novamente, assumimos que as transações não realizam atualizações nos blocos do buffer nem no log enquanto o ponto de verificação está em andamento.

O requisito de que as transações não podem realizar atualizações nos blocos do buffer nem no log durante o ponto de verificação pode ser incômodo, pois o processamento da transação terá de ser interrompido enquanto um ponto de verificação está em andamento. Um **ponto de verificação difuso** é aquele em que as transações têm permissão para realizar atualizações mesmo enquanto os blocos de buffer estão sendo escritos. A seção "Ponto de verificação difuso" descreve os esquemas de ponto de verificação difuso.

Recuperação na partida

Quando o sistema se recupera de uma falha, ele constrói duas listas: a lista de undo consiste em transações a serem desfeitas, e a lista de redo consiste em transações a serem refeitas.

O sistema constrói as duas listas da seguinte maneira. Inicialmente, elas estão vazias. O sistema varre o log de trás para a frente, examinando cada registro até encontrar o primeiro registro <checkpoint>:

- Para cada registro encontrado na forma <T_i ccommit>, ele acrescenta T_i à lista de redo.
- Para cada registro encontrado na forma <T_i start>, se T_i não está na lista de redo, então acrescenta T_i na lista de undo.

Quando o sistema tiver examinado todos os registros de log apropriados, ele verifica a lista L no registro do ponto de verificação. Para cada transação T_i em L, se T_i não estiver na lista de redo, então ele acrescenta T_i à lista de undo.

Quando a lista de redo e a lista de undo tiverem sido construídas, a recuperação prossegue da seguinte maneira:

1. O sistema varre novamente o log do registro mais recente para trás e realiza um undo para cada registro de log que pertence à transação T_i na lista de undo. Os registros de log das transações na lista de redo são ignorados nessa fase. A varredura termina quando os registros <T_i start> tiverem sido encontrados para cada transação T_i na lista de undo.
2. O sistema localiza o registro <checkpoint L> mais recente no log. Observe que essa etapa pode envolver a varredura do log para a frente, se o registro do ponto de verificação foi passado na etapa 1.
3. O sistema varre o log para a frente a partir do registro <checkpoint L> mais recente e realiza redo para cada registro de log que pertence a uma transação T_i que está na lista de redo. Ele ignora os registros de log das transações na lista de undo nessa frase.

É importante, na etapa 1, processar o log de trás para a frente, a fim de assegurar que o estado resultante do banco de dados esteja correto.

Após o sistema ter desfeito todas as transações na lista de undo, ele refaz essas transações na lista de redo. É importante, nesse caso, processar o log de forma direta. Quando o processo de recuperação tiver sido concluído, o processamento da transação retoma.

É importante desfazer a transação na lista de undo antes de refazer as transações na lista de redo, usando o algoritmo das Etapas de 1 a 3; caso contrário, poderá haver um problema. Suponha que o item de dados A inicialmente tenha o valor 10. Suponha que uma transação T_1 atualizasse o item de dados A para 20 e abortasse; o rollback da transação

restauraria A com o valor 10. Suponha que outra transação T_j, em seguida, atualizasse o item de dados A para 30 e confirmasse, e depois o sistema falhasse. O estado do log no momento da falha é

$$<T_i, A, 10, 20>$$
$$<T_j, A, 10, 30>$$
$$<T_j \texttt{ commit}>$$

Se a passada de redo for realizada primeiro, A será definido como 30; depois, na passada de undo, A será definido como 10, o que está errado. O valor final de Q deverá ser 30, que podemos garantir realizando o undo antes do redo.

Gerenciamento de buffer

Nesta seção, consideramos diversos detalhes sutis essenciais para a implementação de um esquema de recuperação de falhas que garanta a coerência dos dados e imponha uma quantidade mínima de sobrecarga nas interações com o banco de dados.

Buffering de registro de log

Até aqui, consideramos que cada registro de log é enviado ao armazenamento estável no momento em que foi criado. Essa suposição impõe uma alta sobrecarga na execução do sistema por vários motivos: normalmente, a saída para armazenamento estável está em unidades de blocos. Assim, a saída de cada registro de log se traduz para uma saída muito maior no nível físico. Além do mais, como vimos na seção "Implementação do armazenamento estável", a saída de um bloco para o armazenamento estável pode envolver várias operações de saída no nível físico.

O custo para enviar um bloco para o armazenamento estável é suficientemente alto que se deseja enviar vários registros de log ao mesmo tempo. Para isso, escrevemos registros de log em um buffer na memória principal, onde permanecem temporariamente até que sejam enviados para o armazenamento estável. Vários registros de log podem ser reunidos no buffer de log e enviados para o armazenamento estável em uma única operação de saída. A ordem dos registros de log no armazenamento estável precisa ser exatamente a mesma em que foram escritos no buffer de log.

Como resultado do buffer de log, um registro de log pode residir apenas na memória principal (armazenamento volátil) por um tempo considerável antes de ser enviado para o armazenamento estável. Como esses registros de log são perdidos se o sistema falhar, temos de impor requisitos adicionais sobre as técnicas de recuperação para garantir a atomicidade da transação:

- A transação T_i entra no estão commit depois que o registro de log $<T_i \texttt{ commit}>$ tiver sido enviado para o armazenamento estável.
- Antes que o registro de log $<T_i \texttt{ commit}>$ possa ser enviado para o armazenamento estável, todos os registros de log pertencentes à transação T_i precisam ter sido enviados ao armazenamento estável.
- Antes que um bloco de dados na memória principal possa ser enviado para o banco de dados (no armazenamento não volátil), todos os registros de log pertencentes aos dados nesse bloco precisam ter sido enviados ao armazenamento estável.

Essa regra é denominada regra do **logging de escrita antecipada** (WAL – Write-Ahead Logging). (Estritamente falando, a regra WAL requer apenas que a informação de undo no log tenha sido enviada ao armazenamento estável e permite que a informação de redo seja escrita mais tarde. A diferença é relevante nos sistemas em que as informações de undo e de redo são armazenadas em registros de log separados.) As três regras indicam situações em que certos registros de log *precisam* ter sido enviados ao armazenamento estável. Não há problema resultante da saída dos registros de log *antes* do necessário. Assim, quando o sistema acha necessário enviar um registro de log para o armazenamento estável, ele envia um bloco inteiro de registros de log, se houver registros de log suficientes na memória principal para preencher um bloco. Se não houver registros de log suficientes para preencher o bloco, todos os registros de log na memória principal são combinados em um bloco parcialmente cheio e enviados ao armazenamento estável.

A escrita do log em buffer para o disco às vezes é conhecida como **forçar o log**.

Buffering de banco de dados

Na seção "Estrutura de armazenamento", descrevemos o uso de uma hierarquia de armazenamento em dois níveis. O sistema armazena o banco de dados no armazenamento não volátil (disco) e leva blocos de dados para a memória principal conforme a necessidade. Como a memória principal normalmente é muito menor do que o banco de dados inteiro, pode ser necessário escrever sobre um bloco B_1 na memória principal quando outro bloco B_2 precisa ser trazido para a memória. Se B_1 tiver sido modificado, B_1 precisa ser enviado antes da entrada de B_2. Conforme discutimos na seção "Gerenciador de buffer" do Capítulo 11, essa hierarquia de armazenamento é o conceito padrão de *memória virtual* do sistema operacional.

As regras para o envio dos registros de log limitam a liberdade do sistema a blocos de saída de dados. Se a entrada do bloco B_2 fizer com que o bloco B_1 seja escolhido para

saída, todos os registros de log pertencentes aos dados em B_1 precisam ser enviados ao armazenamento estável antes que B_1 seja enviado. Assim, a seqüência de ações pelo sistema seria:

- Enviar registros de log ao armazenamento estável até que todos os registros de log pertencentes ao bloco B_1 tenham sido enviados.
- Enviar bloco B_1 para o disco.
- Trazer bloco B_2 do disco para a memória principal.

É importante que nenhuma escrita no bloco B_1 esteja em andamento enquanto o sistema executa essa seqüência de ações. Podemos garantir isso usando um meio especial de bloqueio: antes que a transação realize uma escrita sobre um item de dados, ela precisa adquirir um bloqueio exclusivo sobre o bloco em que o item de dados reside. O bloqueio pode ser liberado imediatamente após a atualização ter sido realizada. Antes que um bloco seja enviado, o sistema obtém um bloqueio exclusivo sobre o bloco, para garantir que nenhuma transação está atualizando o bloco. Ele libera o bloqueio quando a saída do bloco tiver completado. Os bloqueios mantidos por uma curta duração normalmente são chamados de **latches**. Os latches são tratados como sendo distintos dos bloqueios usados pelo sistema de controle de concorrência. Como resultado, eles podem ser liberados sem considerar qualquer protocolo de bloqueio, como o bloqueio em duas fases, exigido pelo sistema de controle de concorrência.

Para ilustrar a necessidade de um requisito de logging de escrita antecipada, considere nosso exemplo de banco com as transações T_0 e T_1. Suponha que o estado do log seja

<T_0 start>
<T_0, A, 1000, 950>

e essa transação T_0 emita um read(B). Considere que o bloco em que B reside não esteja na memória principal e que a memória principal esteja cheia. Suponha que o bloco em que A reside seja escolhido para ser enviado ao disco. Se o sistema enviar esse bloco ao disco e depois houver uma falha, os valores no banco de dados para as contas A, B e C são $950, $2.000 e $700, respectivamente. O estado desse banco de dados está inconsistente. Porém, devido aos requisitos WAL, o registro de log

<T_0, A, 1000, 950>

precisa ser enviado ao armazenamento estável antes da saída do bloco em que A reside. O sistema pode usar o registro de log durante a recuperação para trazer o banco de dados de volta a um estado consistente.

Papel do sistema operacional no gerenciamento de buffer

Podemos gerenciar o buffer do banco de dados usando uma das duas técnicas:

1. O sistema de banco de dados reserva parte da memória principal para servir como um buffer que ele gerencia, em vez do sistema operacional. O sistema de banco de dados gerencia a transferência de bloco de dados de acordo com os requisitos na seção anterior. Essa técnica tem a desvantagem de limitar a flexibilidade no uso da memória principal. O buffer precisa ser mantido pequeno o suficiente para que outras aplicações tenham memória principal suficiente à disposição para suas necessidades. Porém, mesmo quando as outras aplicações não estiverem sendo executadas, o banco de dados não será capaz de utilizar toda a memória disponível. De modo semelhante, as aplicações não de banco de dados podem não usar aquela parte da memória principal reservada para o buffer de banco de dados, mesmo que algumas das páginas no buffer de banco de dados não estejam sendo usadas.
2. O sistema de banco de dados implementa seu buffer dentro da memória virtual fornecida pelo sistema operacional. Como o sistema operacional sabe a respeito dos requisitos de memória de todos os processos no sistema, o ideal é que ele esteja encarregado de decidir quais blocos de buffer devem ter a saída forçada para o disco e quando. No entanto, para garantir os requisitos de logging de escrita antecipada na seção "Buffering de registro de log", o próprio sistema operacional não deverá escrever as páginas do buffer de banco de dados, mas deve solicitar que o sistema de banco de dados force a saída para os blocos de buffer. O sistema de banco de dados, por sua vez, forçaria a saída dos blocos de buffer no banco de dados, após escrever os registros de log relevantes para o armazenamento estável.

   Infelizmente, quase todos os sistemas operacionais da geração atual retêm controle completo da memória virtual. O sistema operacional reserva espaço em disco para armazenar páginas da memória virtual que não estão atualmente na memória principal; esse espaço é denominado **espaço de swap** (ou troca). Se o sistema operacional decidir enviar um bloco B_X, esse bloco é enviado para o espaço de swap no disco e não há como o sistema de banco de dados obter controle da saída dos blocos de buffer.

   Portanto, se o buffer do banco de dados estiver na memória virtual, as transferências entre os arquivos de banco de dados e o buffer na memória virtual

precisam ser gerenciadas pelo sistema de banco de dados, que impõe os requisitos de logging de escrita antecipada que discutimos.

Essa técnica pode resultar em uma saída extra de dados para o disco. Se um bloco B_X for enviado pelo sistema operacional, esse bloco não é enviado ao banco de dados. Em vez disso, ele é enviado ao espaço de swap para a memória virtual do sistema operacional. Quando o sistema de banco de dados precisar enviar B_X, o sistema operacional pode primeiro precisar enviar B_X a partir do seu espaço de swap. Assim, em vez de uma única saída de B_X, pode haver duas saídas de B_X (uma pelo sistema operacional e uma pelo sistema de banco de dados) e uma entrada extra de B_X.

Embora as duas técnicas sofram de algumas desvantagens, uma ou outra precisa ser escolhida, a menos que o sistema operacional seja projetado para dar suporte aos requisitos do logging de banco de dados. Somente alguns sistemas operacionais atuais, como o sistema operacional Mach, admitem esses requisitos.

Falha com perda de armazenamento não volátil

Até agora, consideramos apenas o caso em que uma falha resulta na perda de informações residindo no armazenamento volátil enquanto o conteúdo do armazenamento não volátil permanece intacto. Embora as falhas em que o conteúdo do armazenamento não volátil é perdido sejam raras, precisamos estar preparados para lidar com esse tipo de falha. Nesta seção, discutimos apenas o armazenamento de disco. Nossas discussões se aplicam também a outros tipos de armazenamento não volátil.

O esquema básico é realizar um **dump** do conteúdo inteiro do banco de dados para o armazenamento estável periodicamente – digamos, uma vez por dia. Por exemplo, podemos fazer o dump do banco de dados para uma ou mais fitas magnéticas. Se houver uma falha que resulte na perda de blocos físicos do banco de dados, o sistema usa o dump mais recente na restauração do banco de dados para um estado consistente anterior. Quando essa restauração tiver sido realizada, o sistema utiliza o log para trazer o sistema de banco de dados ao estado consistente mais recente.

Mais precisamente, nenhuma transação pode estar ativa durante o procedimento de dump e um procedimento semelhante ao ponto de verificação precisa acontecer:

1. Enviar todos os registros de log atualmente residindo na memória principal para o armazenamento estável.
2. Enviar todos os blocos de buffer para o disco.
3. Copiar o conteúdo do banco de dados para o armazenamento estável.
4. Enviar um registro de log <dump> para o armazenamento estável.

As etapas 1, 2 e 4 correspondem às três etapas usadas para pontos de verificação na seção "Pontos de verificação (Check points)".

Para recuperar-se da perda de armazenamento não volátil, o sistema restaura o banco de dados ao disco usando o dump mais recente. Depois, ele consulta o log e refaz todas as transações que foram confirmadas desde o dump ocorrido mais recentemente. Observe que nenhuma operação undo precisa ser executada.

Um dump do conteúdo do banco de dados também é conhecido como **dump de arquivamento**, pois podemos arquivar os dumps e usá-los mais tarde para examinar os estados antigos do banco de dados. Os dumps de um banco de dados e o ponto de verificação dos buffers são semelhantes.

O procedimento de dump simples descrito aqui é dispendioso pelos dois motivos a seguir. Primeiro, o banco de dados inteiro precisa ser copiado para o armazenamento estável, resultando em uma transferência de dados considerável. Segundo, como o processamento de transação é interrompido durante o procedimento de dump, ciclos de CPU são desperdiçados. Os esquemas de **dump difuso** foram desenvolvidos para permitir que transações estejam ativas enquanto o dump está em andamento. Eles são semelhantes aos esquemas de ponto de verificação difuso; veja mais detalhes nas notas bibliográficas.

Técnicas de recuperação avançadas**

As técnicas de recuperação descritas na seção "Recuperação com transações concorrentes" exigem que, quando uma transação atualizar um item de dados, nenhuma outra transação poderá atualizar o mesmo item de dados até que a primeira seja confirmada ou revertida. Garantimos a condição usando o bloqueio estrito em duas fases. Embora o bloqueio estrito em duas fases seja aceitável para os registros nas relações, conforme discutimos na seção "Concorrência em estruturas de índice" do Capítulo 16, ele causa uma diminuição significativa na concorrência quando aplicado a certas estruturas especializadas, como páginas de índice de árvore B$^+$.

Para aumentar a concorrência, podemos usar o algoritmo de controle de concorrência de árvore B$^+$ descrito na seção "Concorrência em estruturas de índice" do Capítulo 16 para permitir que os bloqueios sejam liberados mais cedo, não em duas fases. Porém, como resultado, as técnicas de

recuperação da seção "Recuperação com transações concorrentes" se tornarão inaplicáveis. Foram propostas várias técnicas de recuperação alternativas, aplicáveis mesmo com a liberação de bloqueio antecipada. Esses esquemas podem ser usados em diversas aplicações, e não apenas para a recuperação de árvores B+. Primeiro, descrevemos um esquema de recuperação avançado dando suporte à liberação de bloqueio antecipada. Depois, esboçamos o esquema de recuperação ARIES, que é bastante usado no setor. ARIES é mais complexo do que nosso esquema de recuperação avançado, mas incorpora diversas otimizações para minimizar o tempo de recuperação e oferece uma série de recursos úteis.

Logging de undo lógico

Para operações em que os bloqueios são liberados mais cedo, não podemos realizar as ações de undo simplesmente escrevendo de volta o valor antigo dos itens de dados. Considere a transação T_i que insere uma entrada em uma árvore B+ e, seguindo o protocolo de controle de concorrência da árvore B+, libera alguns bloqueios depois que a operação de inserção terminar, mas antes que a transação seja confirmada. Depois que os bloqueios são liberados, outras transações podem realizar outras inserções ou exclusões, causando assim outras mudanças nos nós da árvore B+.

Embora a operação libere alguns bloqueios mais cedo, ela precisa reter bloqueios suficientes para garantir que nenhuma outra transação tenha permissão para executar qualquer operação em conflito (como ler o valor inserido ou excluir o valor inserido). Por esse motivo, o protocolo de controle de concorrência da árvore B+ na seção "Concorrência em estruturas de índice" do Capítulo 16 mantém bloqueios sobre o nível de folha da árvore B+ até o final da transação.

Agora, vamos considerar como realizar o rollback da transação. Se o **undo físico** for usado, ou seja, se os valores antigos dos nós internos da árvore B+ (antes que a operação de inserção fosse executada) forem escritos de volta durante o rollback da transação, algumas das atualizações realizadas pelas operações posteriores de inserção ou exclusão, executadas por outras transações, poderiam ser perdidas. Em vez disso, a operação de inserção precisa ser desfeita por um **undo lógico** – ou seja, neste caso, pela execução de uma operação de exclusão.

Portanto, quando a operação de inserção termina, antes que libere quaisquer bloqueios, ela escreve um registro de log $<T_i, O_j,$ `operation-end`, $U>$, onde U indica a informação de undo e O_j indica um identificador exclusivo para (a instância de) a operação. Por exemplo, se a operação inserisse uma entrada em uma árvore B+, a informação de undo U indicaria que a operação de exclusão deve ser realizada e identificaria a árvore B+ e o que excluir da árvore. Esse logging de informações sobre operações é chamado de **logging lógico**. Ao contrário, o logging da informação de valor antigo e valor novo é chamado de **logging físico**, e os registros de log correspondentes são chamados **registros de log físico**.

As operações de inserção e exclusão são exemplos de uma classe de operações que exigem operações de undo lógico, pois liberam bloqueios mais cedo; chamamos essas operações de **operações lógicas**. Antes que uma operação lógica comece, ela escreve um registro de log $<T_i, O_j,$ `operation-begin`, onde O_j é o identificador exclusivo para a operação. Enquanto o sistema está executando a operação, ele realiza o logging físico da forma normal para todas as atualizações realizadas pela operação. Assim, a informação normal do valor antigo e do valor novo é escrita para cada atualização. Quando a operação termina, ela escreve um registro de log operation-end, conforme descrevemos anteriormente.

Rollback de transação

Primeiro, considere o rollback de transação durante a operação normal (ou seja, não durante a recuperação de uma falha do sistema). O sistema varre o log de trás para a frente e usa os registros de log pertencentes à transação para restaurar os valores antigos dos itens de dados. Porém, ao contrário do rollback na operação normal, o rollback em nosso esquema de recuperação avançado escreve registros de log especiais somente de redo, na forma $<T_i, X_j, V>$, contendo o valor V sendo restaurado ao item de dados X_j durante o rollback. Esses registros de log às vezes são denominados **registros de log de compensação**. Esses registros não precisam desfazer informações, pois nunca precisaremos desfazer tal operação de undo.

Sempre que o sistema encontra um registro de log $<T_i, O_j,$ `operation-end`, $U>$, ele toma ações especiais:

1. Reverte a operação, usando a informação de undo U no registro de log. Ele registra em log as atualizações realizadas durante o rollback da operação exatamente como as atualizações realizadas quando a operação foi executada inicialmente. Em outras palavras, o sistema registra informações de undo físico para as atualizações realizadas durante o rollback, em vez de usar os registros de log de compensação. Isso porque uma falha pode ocorrer enquanto um undo lógico está em andamento e, na recuperação, o sistema precisa completar o undo lógico. Para isso, a recuperação na partida desfará os efeitos parciais do undo inicial, usando a infor-

mação do undo físico, e depois realizará o undo lógico novamente, conforme veremos na seção "Recuperação na partida".

Ao final do rollback da operação, em vez de gerar um registrador de log <T_i, O_j, operation-end, U>, o sistema de banco de dados gera um registro de log <T_i, O_j, `operation-abort`>.

2. Quando a varredura inversa do log continua, o sistema pula todos os registros de log da transação até encontrar o registro de log <T_i, O_j, `operation-begin`>. Depois de encontrar o registro de log `operation-begin`, ele processa os registros de log da transação pelo modo normal novamente.

Observe que pular os registros de log físicos quando o registro de log operation-end é encontrado durante o rollback assegura que os valores antigos no registro de log físico não sejam usados para o rollback, quando a operação for concluída.

Se o sistema encontrar um registro <T_i, O_j, `operation-abort`>, ele pula todos os registros anteriores até encontrar o registro <T_i, O_j, `operation-begin`>. Esses registros de log anteriores precisam ser pulados para impedir o rollback múltiplo da mesma operação, caso tenha havido uma falha durante um rollback anterior e a transação já tivesse sido parcialmente revertida. Quando a transação T_i tiver sido revertida, o sistema acrescentará um registro <T_i `abort`> ao log.

Se houver falhas enquanto uma operação lógica estiver em andamento, o registro de log `operation-end` para a operação não será encontrado quando a transação for revertida. Porém, para cada atualização realizada pela operação, informações de undo – na forma do valor antigo nos registros de log físicos – estão disponíveis no log. Os registros de log físicos serão usados para reverter a operação incompleta.

Pontos de verificação

O ponto de verificação é realizado conforme descrito na seção "Recuperação com transações concorrentes". O sistema suspende temporariamente as atualizações ao banco de dados e executa estas ações:

1. Envia para o armazenamento estável todos os registros de log atualmente residindo na memória principal.
2. Envia para o disco todos os blocos de buffer modificados.
3. Envia para o armazenamento estável um registro de log <`checkpoint` L>, onde L é uma lista de todas as transações ativas.

Recuperação na partida

As ações de recuperação, quando o sistema de banco de dados é reiniciado após uma falha, ocorrem em duas fases:

1. Na **fase de redo**, o sistema repete as atualizações de *todas* as transações varrendo o log para a frente a partir do último ponto de verificação. Os registros de log que são repetidos incluem os registros de log para transações que foram revertidas antes da falha do sistema e aquelas que não tinham sido confirmadas quando ocorreu a falha do sistema. Os registros são os registros de log normais, na forma <T_i, X_j, V_1, V_2>, bem como os registros de log especiais na forma <T_i, X_j, V_2>; o valor V_2 é escrito no item de dados X_j de qualquer forma. Essa fase também determina todas as transações que estão ou na lista de transação no registro do ponto de verificação, ou iniciadas depois, mas não tinham um <T_i `abort`> ou um registro <T_i `commit`> no log. Todas essas transações precisam ser revertidas, e o sistema coloca seus identificadores de transação em uma lista de undo.
2. Na **fase de undo**, o sistema reverte todas as transações na lista de undo. Ele realiza o rollback varrendo o log para trás a partir do final. Sempre que encontra um registro de log pertencente a uma transação na lista de undo, ele realiza ações de undo exatamente como se o registro de log tivesse sido encontrado durante o rollback de uma transação que falhou. Assim, os registros de log de uma transação antes de um registro `operation-end`, mas depois do registro `operation-begin` correspondente, são ignorados.

   Quando o sistema encontra um registro de log <T_i `start`> para uma transação T_i na lista de undo, ele escreve um registro de log <T_i `abort`> no log. A varredura do log para quando o sistema tiver encontrado registros de log <T_i `start`> para todas as transações na lista de undo.

A fase de redo da recuperação na partida repete cada registro de log físico desde o registro de ponto de verificação mais recente. Em outras palavras, essa fase da recuperação na partida repete todas as ações de atualização que foram executadas após o ponto de verificação e cujos registros de log alcançaram o log estável. As ações incluem ações de transações incompletas e as ações executadas para reverter transações que falharam. As ações são repetidas na mesma ordem em que foram executadas; daí esse processo ser chamado **histórico repetitivo**. O histórico repetitivo simplifica muito os esquemas de recuperação.

Observe que, se uma operação undo esteve em andamento quando houve a falha do sistema, os registros de log

físicos escritos durante a operação undo seriam encontrados, e a própria operação undo parcial seria desfeita com base nesses registros de log físicos. Depois disso, o registro `operation-end` da operação original seria encontrado durante a recuperação, e a operação undo seria executada novamente.

Ponto de verificação difuso

A técnica de ponto de verificação descrita na seção "Pontos de verificação" requer que todas as atualizações no banco de dados sejam suspensas temporariamente enquanto o ponto de verificação está em andamento. Se o número de páginas no buffer for grande, um ponto de verificação pode levar muito tempo para terminar, o que pode resultar em uma interrupção inaceitável no processamento de transações.

Para evitar tais interrupções, a técnica de ponto de verificação pode ser modificada para permitir que as atualizações comecem quando o registro `checkpoint` tiver sido escrito, mas antes que os blocos de buffer modificados sejam gravados em disco. O ponto de verificação gerado dessa forma é um **ponto de verificação difuso**.

Como as páginas são enviadas ao disco somente depois que o registro `checkpoint` tiver sido escrito, é possível que o sistema falhe antes que todas as páginas sejam escritas. Assim, um ponto de verificação no disco pode estar incompleto. Um modo de tratar de pontos de verificação incompletos é este: o local no log do registro de ponto de verificação do último ponto de verificação completado é armazenado em uma posição fixa no disco. O último ponto de verificação. O sistema não atualiza essa informação quando escrever o registro `checkpoint` . Em vez disso, antes de escrever o registro `checkpoint`, ele cria uma lista de todos os blocos de buffer modificados. A informação do último ponto de verificação só é atualizada depois que todos os blocos de buffer na lista de blocos de buffer modificados tenham sido enviados ao disco.

Mesmo com o ponto de verificação difuso, um bloco de buffer não deve ser atualizado enquanto está sendo enviado ao disco, embora outros blocos de buffer possam ser atualizados simultaneamente. O protocolo de log de escrita antecipada precisa ser seguido de modo que os registros de log (undo) pertencentes a um bloco estejam no armazenamento estável antes que o bloco seja enviado.

Observe que, em nosso esquema, o logging lógico é usado somente para fins de undo, enquanto o logging físico é usado para fins de redo e undo. Existem esquemas de recuperação que usam o logging lógico para fins de redo. Para realizar o redo lógico, o estado do banco de dados no disco precisa ter **consistência de operação**, ou seja, não deve ter efeitos parciais de qualquer operação. É difícil garantir a consistência de operação do banco de dados no disco se uma operação puder afetar mais de uma página, pois não é possível escrever duas ou mais páginas de forma indivisível. Portanto, o logging de redo lógico normalmente é restrito apenas a operações que afetam uma página do sistema; veremos como lidar com esses redos lógicos na próxima seção. Ao contrário, os undos lógicos são realizados sobre um estado de banco de dados com consistência de operação, alcançado pelo histórico repetitivo e depois realizando o undo físico das operações parcialmente completadas.

ARIES

O que há de mais moderno em métodos de recuperação é ilustrado pelo método de recuperação ARIES. A técnica de recuperação avançada que descrevemos é modelada pelo ARIES, mas foi bastante simplificada para destacar os principais conceitos e torná-la mais fácil de entender. Ao contrário, ARIES usa diversas técnicas para reduzir o tempo gasto para a recuperação e reduzir os overheads do ponto de verificação. Em particular, ARIES é capaz de evitar refazer muitas operações em log que já foram aplicadas e reduzir a quantidade de informações registradas em log. O preço pago é a maior complexidade; os benefícios compensam o preço.

As principais diferenças entre ARIES e nosso algoritmo de recuperação avançado são que ARIES:

1. Utiliza um **número de seqüência de log** (LSN – Log Sequence Number) para identificar registros de log e utiliza LSNs nas páginas do banco de dados para identificar quais operações foram aplicadas a uma página do banco de dados.
2. Admite operações de **redo fisiológico**, que são físicos porque a página afetada é identificada fisicamente, mas podem ser lógicos dentro da página. Por exemplo, a exclusão de um registro de uma página pode resultar em muitos outros registros na página sendo deslocados, se for utilizada uma estrutura de página em slots. Com o logging de redo físico, todos os bytes da página afetada pelo deslocamento de registros precisam ser registrados em log. Com o logging fisiológico, a operação de exclusão pode ser registrada em log, resultando em um registro de log muito menor. O redo da operação de exclusão excluiria o registro e deslocaria outros registros conforme a necessidade.
3. Utiliza uma **tabela de página suja** para minimizar redos desnecessários durante a recuperação. As páginas sujas são aquelas que foram atualizadas na memória e a versão de disco não está atualizada.
4. Utiliza um esquema de ponto de verificação difuso, que registra apenas informações sobre as páginas

sujas e informações associadas, e nem sequer exige a escrita de páginas sujas em disco. Ele esvazia páginas sujas em segundo plano, continuamente, em vez de escrevê-las durante os pontos de verificação.

No restante desta seção, oferecemos uma visão geral do ARIES. As notas bibliográficas listam referências que oferecem uma descrição completa do ARIES.

Estruturas de dados

Cada registro de log no ARIES possui um **número de sequência de log** (LSN – Log Sequence Number) que identifica o registro de forma exclusiva. O número é, em conceito, apenas um identificador lógico cujo valor é maior para registros de log que ocorrem mais adiante no log. Na prática, o LSN é gerado de modo que também possa ser usado para localizar o registro de log no disco. Normalmente, ARIES divide um log em vários arquivos de log, cada qual com um número de arquivo. Quando um arquivo de log cresce até algum limite, ARIES acrescenta outros registros de log a um novo arquivo de log; o novo arquivo de log possui um número de arquivo 1 a mais que o arquivo de log anterior. O LSN, então, consiste em um número de arquivo e um deslocamento dentro do arquivo.

Cada página também mantém um identificador chamado de **PageLSN**. Sempre que uma operação (seja ela física ou lógica) ocorre em uma página, a operação armazena o LSN do seu registro de log no campo PageLSN da página. Durante a fase de redo da recuperação, quaisquer registros de log com LSN menor ou igual ao PageLSN de uma página não devem ser executados na página, pois suas ações já são refletidas na página. Em combinação com um esquema para registrar PageLSNs como parte do ponto de verificação, que apresentamos mais adiante, ARIES pode evitar até mesmo a leitura de muitas páginas para as quais as operações em log já estão refletidas no disco. Desse modo, o tempo de recuperação é reduzido de modo significativo.

O PageLSN é essencial para garantir a idempotência na presença das operações de redo fisiológicas, pois a reaplicação de um redo fisiológico que já foi aplicado a uma página poderia causar mudanças incorretas em uma página.

As páginas não devem ser esvaziadas para o disco enquanto uma atualização está em andamento, pois as operações fisiológicas não podem ser refeitas em um estado parcialmente atualizado da página no disco. Portanto, ARIES utiliza latches nas páginas de buffer para evitar que eles sejam gravados em disco enquanto estão sendo atualizados. Ele só libera o latch da página de buffer depois que a atualização for concluída e o registro de log para a atualização tiver sido escrito no log.

Cada registro também contém o LSN do registro de log anterior da mesma transação. Esse valor, armazenado no campo PrevLSN, permite que os registros de log de uma transação sejam apanhados ao contrário, sem a leitura do log inteiro. Existem registros de log especiais, somente de redo, gerados durante o rollback da transação, chamados **registros de log de compensação** (CLRs – Compensation Log Records) no ARIES. Estes têm a mesma finalidade dos registros de log somente de redo no nosso esquema de recuperação avançado. Além disso, os CLRs têm a função dos registros de log operation-abort em nosso esquema. Os CLRs têm um campo extra, chamado UndoNextLSN, que registra o LSN do log que precisa ser desfeito em seguida, quando a transação está sendo revertida. Esse campo tem a mesma finalidade do identificador de operação no registro de log operation-abort em nosso esquema, o que ajuda a saltar os registros de log que já foram revertidos. A **DirtyPageTable** contém uma lista das páginas que foram atualizadas no buffer do banco de dados. Para cada página, ela armazena o PageLSN e um campo chamado RecLSN, que ajuda a identificar registros de log que já foram aplicados à versão da página no disco. Quando uma página é inserida na DirtyPageTable (quando ela é modificada inicialmente no pool de buffer), o valor de RecLSN é definido como o final atual do log. Sempre que a página é esvaziada para o disco, ela é removida da DirtyPageTable.

Um **registro de log de ponto de verificação** contém a DirtyPageTable e uma lista de transações ativas. Para cada transação, o registro de log do ponto de verificação também anota LastLSN, o LSN do último registro de log escrito pela transação. Uma posição fixa no disco também anota o LSN do último registro de log do ponto de verificação (completo).

Algoritmo de recuperação

O ARIES se recupera de uma falha do sistema em três passadas.

- **Passada de análise**: essa passada determina quais transações desfazer, quais páginas estavam sujas no momento da falha e o LSN do qual a passada de redo deve começar.
- **Passada de redo**: essa passada começa com uma posição determinada durante a análise e realiza um histórico de redo, repetitivo, para trazer o banco de dados a um estado em que estava antes da falha.
- **Passada de undo**: essa passada reverte todas as transações que estavam incompletas no momento da falha.

Passada de análise: A passada de análise encontra o último registro de log de ponto de verificação completo e lê na DirtyPageTable desse registro. Depois, ela define Re-

doLSN com o menor dos RecLSNs das páginas na DirtyPageTable. Se não houver páginas sujas, ela define RedoLSN com o LSN do registro de log do ponto de verificação. A passada de redo inicia sua varredura do log a partir do RedoLSN. Todos os registros de log antes desse ponto já foram aplicados às páginas do banco de dados no disco. A passada de análise inicialmente define a lista de transações a serem desfeitas, a lista de undo, até a lista de transações no registro de log do ponto de verificação. A passada de análise também lê do registro de log do ponto de verificação os LSNs do último registro de log para cada transação na lista de undo.

A passada de análise continua varrendo para a frente a partir do ponto de verificação. Sempre que encontrar um registro de log para uma transação que não esteja na lista de undo, ela acrescenta a transação na lista de undo. Sempre que encontrar um registro de log de fim de transação, ela exclui a transação da lista de undo. Todas as transações deixadas na lista de undo no final da análise precisam ser revertidas mais tarde, na passada de undo. A passada de análise também registra o último registro de cada transação na lista de undo, que é usado na passada de undo.

A passada de análise também atualiza a DirtyPageTable sempre que encontrar um registro de log para uma atualização em uma página. Se a página não estiver na DirtyPageTable, a passada de análise a acrescenta na DirtyPageTable e define o RecLSN da página como o LSN do registro de log.

Passada de redo: A passada de redo repete o histórico substituindo cada ação que ainda não tenha sido refletida na página do disco. A passada de redo varre o log para a frente a partir de RedoLSN. Sempre que encontrar um registro de log de atualização, ele realiza esta ação:

1. Se a página não estiver na DirtyPageTable ou se o LSN do registro de log de atualização for menor que o RecLSN da página em DirtyPageTable, então a passada de redo pula o registro de log.
2. Caso contrário, a passada de redo apanha a página do disco e, se o PageLSN for menor que o LSN do registro de logo, refaz o registro de log.

Observe que, se um dos testes for negativo, então os efeitos do registro de log já apareceram na página. Se o primeiro teste for negativo, nem sequer será necessário apanhar a página do disco.

Passada de undo e rollback da transação: A passada de undo é relativamente simples. Ela realiza uma varredura inversa do log, desfazendo todas as transações na lista de undo. Se um CLR for encontrado, ela usa o campo UndoNextLSN para pular os registros de log que já foram revertidos. Caso contrário, ela usa o campo PrevLSN do registro de log para encontrar o próximo registro de log a ser desfeito.

Sempre que um registro de log de atualização é usado para realizar um undo (seja para o rollback da transação durante o processamento normal ou durante a passada de undo na partida), a passada de undo gera um CLR contendo a ação de undo realizada (que precisa ser fisiológica). Ela define o UndoNextLSN do CLR como o valor de PrevLSN do registro de log de atualização.

Outros recursos

Entre outros recursos-chave que o ARIES oferece estão:

- **Independência de recuperação:** algumas páginas podem ser recuperadas independentemente de outras, de modo que possam ser usadas mesmo enquanto outras páginas estão sendo recuperadas. Se algumas páginas de um disco falharem, elas poderão ser recuperadas sem interromper o processamento de transações em outras páginas.
- **Pontos de salvamento:** transações podem registrar pontos de salvamento e podem ser revertidas parcialmente, até um ponto de salvamento. Isso pode ser muito útil para tratamento de impasse, pois as transações podem ser revertidas até um ponto que permite a liberação de bloqueios exigidos e depois são reiniciadas a partir desse ponto.
- **Bloqueio minucioso:** o algoritmo de recuperação ARIES pode ser usado com algoritmos de controle de concorrência de índice, que permitem o bloqueio em índices no nível de tupla, em vez de bloqueio em nível de página, o que melhora significativamente a concorrência.
- **Otimizações de recuperação:** a DirtyPageTable pode ser usada para a pré-busca de páginas durante o redo, em vez de apanhar uma página somente quando o sistema encontrar um registro de log a ser aplicado à página. O redo fora de ordem também é possível: o redo pode ser adiado em uma página sendo apanhada do disco e realizado quando a página for apanhada. Nesse meio tempo, outros registros de log podem continuar sendo processados.

Resumindo, o algoritmo ARIES é o máximo em algoritmo de recuperação, incorporando uma série de otimizações preparadas para melhorar a concorrência, reduzir a sobrecarga do logging e reduzir o tempo de recuperação.

Sistemas de backup remoto

Os sistemas tradicionais de processamento de transação são sistemas centralizados ou cliente-servidor. Esses sistemas são vulneráveis a desastres ambientais, como incêndio,

inundação ou terremotos. Cada vez mais, existe uma necessidade de sistemas de processamento de transação que possam funcionar apesar das falhas do sistema ou dos desastres ambientais. Esses sistemas precisam oferecer **alta disponibilidade**, ou seja, o tempo durante o qual o sistema não pode ser usado precisa ser extremamente pequeno.

Podemos conseguir alta disponibilidade realizando o processamento da transação em um local, chamado **site primário** e ter um site de **backup remoto**, em que todos os dados do site primário são replicados. O site de backup remoto às vezes é chamado de **site secundário**. O site remoto precisa ser mantido sincronizado com o site primário, enquanto as atualizações são realizadas no primário. Alcançamos o sincronismo enviando todos os registros de log para o site de backup remoto. O site de backup remoto precisa estar separado fisicamente do primário – por exemplo, podemos colocá-lo em um estado diferente –, de modo que um desastre no primário não danifique o site de backup remoto. A Figura 17.8 mostra a arquitetura de um sistema de backup remoto.

Quando o site remoto falha, o site de backup remoto assume o processamento. Porém, primeiro, ele realiza a recuperação, usando sua cópia (talvez desatualizada) dos dados do site primário e os registros de log recebidos do site primário. Com efeito, o site de backup remoto está realizando ações de recuperação que teriam sido realizadas no site primário quando este fosse recuperado. Algoritmos de recuperação padrão, com pequenas modificações, podem ser usados para a recuperação no site de backup remoto. Quando a recuperação tiver sido realizada, o site de backup remoto começa a processar as transações.

A disponibilidade aumenta bastante em comparação com um sistema de único site, pois o sistema pode se recuperar mesmo que todos os dados no site primário sejam perdidos. O desempenho de um sistema de backup remoto é melhor do que o desempenho de um sistema distribuído com um commit em duas fases.

Várias questões precisam ser resolvidas no projeto de um sistema de backup remoto:

- **Detecção de falha.** Como nos protocolos de tratamento de falha para o sistema distribuído, é importante que o sistema de backup remoto detecte quando o site primário falhou. A falha das linhas de comunicação pode fazer com que o backup remoto acredite falsamente que o site primário falhou. Para evitar esse problema, mantemos vários enlaces de comunicação com modos de falha independentes entre o site primário e o backup remoto. Por exemplo, além da conexão com a rede, pode haver uma conexão de modem separada por uma linha telefônica, com serviços fornecidos por diferentes empresas de telecomunicação. Essas conexões podem ter backup por intervenção manual pelos operadores, que podem se comunicar pelo sistema telefônico.
- **Transferência de controle.** Quando o site primário falha, o site de backup assume o processamento e se torna o novo site primário. Quando o site primário original se recupera, ele pode desempenhar o papel de backup remoto ou assumir o papel de site primário novamente. De qualquer forma, o antigo site primário precisa receber um log de atualizações executadas pelo site de backup, enquanto o antigo site primário estava inoperante.

  O modo mais simples de transferir o controle é fazendo com que o antigo site primário receba logs de redo do antigo site de backup e acompanhe as atualizações aplicando-as localmente. O antigo site primário pode, então, atuar como site de backup remoto. Se o controle tiver de ser transferido de volta, o antigo site de backup pode fingir ter falhado, fazendo com que o antigo site primário assuma sua função.
- **Tempo de recuperação.** Se o log no backup remoto se tornar muito grande, a recuperação levará muito tempo. O site de backup remoto pode processar periodicamente os registros de log de redo que recebeu e realizar um ponto de verificação, de modo que as partes anteriores do log possam ser excluídas. Como resultado disso, o atraso antes que o backup remoto assuma pode ser bastante reduzido.

  Uma configuração de **reserva a quente** (hot-spare) pode tornar quase instantânea a tomada de comando pelo site de backup. Nessa configuração, o site de backup remoto processa continuamente os registros de log de redo quando eles chegam, aplicando as atualizações

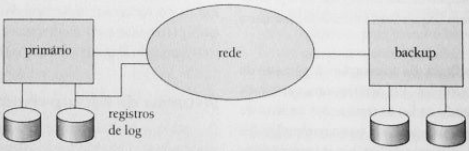


Figura 17.8 Arquitetura do sistema de backup remoto.

localmente. Assim que a falha do site primário é detectada, o site de backup completa a recuperação, revertendo as transações incompletas; em seguida, ele está pronto para processar novas transações.

- **Tempo para confirmar.** Para garantir que as atualizações da transação confirmada sejam duráveis, uma transação não deve ser declarada como confirmada até que seus registros de log tenham alcançado o site de backup. Esse atraso pode resultar em uma espera maior para confirmar uma transação, e alguns sistemas, portanto, permitem menores graus de durabilidade. Os graus de durabilidade podem ser classificados da seguinte forma.
  - **Um seguro.** Uma transação é confirmada assim que seu registro de log de commit é escrito no armazenamento estável no site primário.
    O problema com esse esquema é que as atualizações de uma transação confirmada podem não ter sido feitas no site de backup, quando o site de backup assumir o processamento. Assim, as atualizações podem parecer estar perdidas. Quando o site primário se recuperar, as atualizações perdidas não poderão ser mescladas diretamente, pois as atualizações podem entrar em conflito com atualizações posteriores, realizadas no site de backup. Assim, a intervenção humana pode ser exigida para levar o banco de dados a um estado coerente.
  - **Dois muito seguros.** Uma transação é confirmada assim que seu registro de log de commit é escrito no armazenamento estável no site de backup primário e de backup.
    O problema com esse esquema é que o processamento da transação não pode prosseguir se o site primário ou de backup estiver inoperante. Assim, a disponibilidade é realmente menor do que no caso de único site, embora a probabilidade de perda de dados seja muito menor.
  - **Dois seguros.** Esse esquema é o mesmo que o esquema "dois muito seguros" se os sites primário e de backup estiverem ativos. Se apenas o primário estiver ativo, a transação tem permissão para ser confirmada assim que seu registro de log de commit for escrito no armazenamento estável no site primário.
    Esse esquema oferece melhor disponibilidade do que o esquema "dois muito seguros", enquanto evita o problema de transações perdidas, enfrentado pelo esquema "um seguro". Ele resulta em um commit mais lento do que no esquema "um seguro", mas os benefícios geralmente superam o custo.

Vários sistemas comerciais de disco compartilhado oferecem um nível de tolerância a falhas que é um intermediário entre os sistemas de backup centralizado e remoto. Nesses sistemas comerciais, a falha de uma CPU não resulta na falha do sistema. Em vez disso, outras CPUs assumem e executam a recuperação. As ações de recuperação incluem o rollback de transações sendo executadas na CPU que falhou e a recuperação de bloqueios mantidos por essas transações. Como os dados estão em um disco compartilhado, não há necessidade de transferência de registros de log. Porém, devemos proteger os dados contra falha de disco usando, por exemplo, uma organização de disco RAID.

Um modo alternativo de alcançar alta disponibilidade é usar um banco de dados distribuído, com dados replicados em mais de um site. As transações, então, precisam atualizar todas as réplicas de qualquer item de dados que elas atualizam. Estudamos os bancos de dados distribuídos, incluindo a replicação, no Capítulo 22.

Resumo

- Um sistema de computador, como qualquer outro dispositivo mecânico ou elétrico, está sujeito a falhas. Existem diversas causas para esse tipo de falha, incluindo falha de disco, falta de energia e erros de software. Em cada um desses casos, as informações referentes ao sistema de banco de dados são perdidas.
- Além das falhas do sistema, as transações também podem falhar por vários motivos, como violação de restrições de integridade ou impasses.
- Uma parte integral de um sistema de banco de dados é um esquema de recuperação responsável pela detecção de falhas e para a restauração do banco de dados a um estado que existia antes da ocorrência da falha.
- Os diversos tipos de armazenamento em um computador são o armazenamento volátil, o armazenamento não volátil e o armazenamento estável. Os dados no armazenamento volátil, como na RAM, são perdidos devido a problemas como falhas de disco. Os dados no armazenamento estável nunca são perdidos.
- O armazenamento estável que precisa ser acessível on-line é aproximado aos discos espelhados, ou outras formas de RAID, que oferecem armazenamento de dados redundante. O armazenamento estável offline, ou de arquivamento, pode consistir em várias cópias de fita dos dados armazenados em um local fisicamente seguro.
- No caso de falha, o estado do sistema de banco de dados pode não estar mais consistente; ou seja, ele pode não refletir um estado do mundo que o banco de dados deveria capturar. Para preservar a consistência, exigimos que cada transação seja atômica. É responsabilidade do esquema de recuperação garantir as propriedades de atomicidade e durabilidade.

- Nos esquemas baseados em log, todas as atualizações são registradas em um log, que precisa ser mantido no armazenamento estável.
  - No esquema de modificações adiadas, durante a execução de uma transação, todas as operações `write` são adiadas até que a transação execute um commit parcial, quando o sistema utiliza a informação no log associado à transação na execução das escritas adiadas.
  - No esquema de modificações imediatas, o sistema aplica todas as atualizações diretamente no banco de dados. Se houver uma falha, o sistema usa a informação do log na restauração do estado do sistema a um estado consistente anterior.

  Para reduzir a sobrecarga da pesquisa de transações de log e da reaplicação, podemos usar a técnica de ponto de verificação.
- O processamento da transação é baseado em um modelo de armazenamento em que a memória principal mantém um buffer de log, um buffer de banco de dados e um buffer do sistema. O buffer do sistema mantém páginas do código objeto do sistema e as áreas de trabalho locais das transações.
- A implementação eficiente de um esquema de recuperação exige que o número de escritas no banco de dados e no armazenamento estável seja reduzido. Os registros de log podem ser mantidos inicialmente no buffer de log volátil, mas precisam ser escritos no armazenamento estável quando uma das seguintes condições ocorrer:
  - Antes que o registro de log <T_i `commit`> poder ser enviado ao armazenamento estável, todos os registros de log pertencentes à transação T_i precisam ter sido enviados ao armazenamento estável.
  - Antes que um bloco de dados na memória principal seja enviado ao banco de dados (no armazenamento não volátil), todos os registros de log pertencentes aos dados nesse bloco precisam ter sido enviados ao armazenamento estável.
- Para a recuperação de falhas que resultam na perda de armazenamento não volátil, temos de fazer o dump do conteúdo inteiro do banco de dados para o armazenamento estável periodicamente – digamos, uma vez por dia. Se houver uma falha que resulta na perda de blocos físicos do banco de dados, usamos o dump mais recente na restauração do banco de dados para um estado consistente anterior. Quando essa restauração tiver sido realizada, usamos o log para trazer o sistema de banco de dados ao estado consistente mais recente.
- Técnicas de recuperação avançadas admitem técnicas de bloqueio com alta concorrência, como aquelas usadas para o controle de concorrência da árvore B+. Essas técnicas são baseadas no undo (lógico) e seguem o princípio do histórico repetitivo. Ao recuperar-se da falha do sistema, o sistema realiza uma passada de redo usando o log, seguido por uma passada de redo no log para reverter transações incompletas.
- O esquema de recuperação ARIES é o esquema mais moderno, que admite uma série de recursos para oferecer maior concorrência, reduzir as sobrecargas de logging e minimizar o tempo de recuperação. Ele também é baseado no histórico repetitivo e permite operações de undo lógico. O esquema esvazia páginas continuamente e não precisa esvaziar todas as páginas no momento de um ponto de verificação. Ele usa números de seqüência lógicos (LSNs) para implementar uma série de otimizações que reduzem o tempo gasto para a recuperação.
- Os sistemas de backup remoto oferecem um alto grau de disponibilidade, permitindo que o processamento da transação continue mesmo que o site primário seja destruído por um incêndio, inundação ou terremoto.

Termos de revisão

- Esquema de recuperação
- Classificação de falha
  - Falha de transação
  - Erro lógico
  - Erro do sistema
  - Falha do sistema
  - Falha de transferência de dados
- Suposição falhar-parar
- Falha de disco
- Tipos de armazenamento
  - Armazenamento volátil
  - Armazenamento não volátil
  - Armazenamento estável
- Blocos
  - Blocos físicos
  - Blocos de buffer
- Buffer de disco
- Saída forçada
- Recuperação baseada em log
- Log
- Registros de log
- Registro de log de atualização
- Modificação adiada
- Idempotência
- Modificação imediata
- Modificações não confirmadas
- Pontos de verificação
- Coleta de lixo
- Recuperação com transações concorrentes
  - Rollback de transação
  - Ponto de verificação difuso
  - Recuperação na partida

- Gerenciamento de buffer
- Buffering de registro de log
- Write-Ahead Logging (WAL)
- Log forçado
- Buffering de banco de dados
- Latches
- Sistema operacional e gerenciamento de buffer
- Perda de armazenamento não volátil
- Dump de arquivamento
- Dump difuso
- Técnica de recuperação avançada
  - Undo físico
  - Undo lógico
  - Logging físico
  - Logging lógico
  - Operações lógicas
  - Rollback de transações
  - Pontos de verificação
  - Recuperação na partida
  - Fase de redo
  - Fase de undo
- Histórico repetitivo
- Ponto de verificação difuso
- ARIES
  - Número de seqüência de log (LSN)
  - PageLSN
  - Redo fisiológico
  - Registro de log de compensação (CLR)
  - DirtyPageTable
  - Registro de log de ponto de verificação
- Alta disponibilidade
- Sistemas de backup remoto
  - Site primário
  - Site de backup remoto
  - Site secundário
- Detecção de falha
- Transferência de controle
- Tempo de recuperação
- Configuração hot-spare
- Tempo para confirmar
  - Um seguro
  - Dois muito seguros
  - Dois seguros

Exercícios práticos

17.1 Compare as versões de modificação adiada e imediata do esquema de recuperação baseado em log em termos de facilidade de implementação e custo adicional.

17.2 Quando o sistema se recupera de uma falha (ver seção "Recuperação na partida"), ele constrói uma lista de undo e uma lista de redo. Explique por que os registros de log para as transações na lista de undo precisam ser processados na ordem inversa, enquanto os registros de log para transações na lista de redo são processados em uma direção normal.

17.3 Explique os motivos para a recuperação de transações interativas ser mais difícil de tratar do que a recuperação de transações em lote. Existe algum modo simples de lidar com essa dificuldade? (Dica: considere uma transação de caixa eletrônico em que o dinheiro é sacado.)

17.4 Às vezes, uma transação precisa ser desfeita depois de confirmada, pois foi executada erroneamente, por exemplo, devido à entrada errônea por um caixa de banco.

a. Dê um exemplo para mostrar que o uso do mecanismo de undo de transação normal para desfazer tal transação poderia levar a um estado inconsistente.

b. Um modo de lidar com essa situação é levar o banco de dados inteiro a um estado anterior ao commit da transação errônea (chamada recuperação em um *ponto no tempo*). As transações que foram confirmadas mais tarde têm seus efeitos revertidos com esse esquema.
Sugira uma modificação no mecanismo de recuperação avançado para implementar a recuperação em um ponto no tempo.

c. Transações não confirmadas podem ser reexecutadas logicamente, mas não podem ser reexecutadas usando seus registros de log.

17.5 O logging de atualizações não é feito explicitamente nas linguagens de programação persistentes. Descreva como as proteções de acesso à página fornecidas pelos sistemas operacionais modernos podem ser usadas para criar imagens antes e depois das páginas que são atualizadas. (Dica: ver Exercício prático 16.9.)

17.6 ARIES considera que existe espaço em cada página para um LSN. Ao lidar com objetos grandes, que se espalham por várias páginas, como arquivos de sistema operacional, uma página inteira pode ser usada por um objeto, sem deixar espaço para o LSN. Sugira uma técnica para lidar com tal situação; sua técnica precisa dar suporte a redos físicos, mas não precisa admitir redos fisiológicos.

Exercícios

17.7 Explique a diferença entre os três tipos de armazenamento – volátil, não volátil e estável – em termos de custo de E/S.

17.8 O armazenamento estável não pode ser implementado.
   a. Explique por que ele não pode ser.
   b. Explique como os sistemas de banco de dados lidam com esse problema.

17.9 Considere que a modificação imediata é usada em um sistema. Mostre, por meio de um exemplo, como poderia ficar o estado de um banco de dados inconsistente se os registros de log para uma transação não fossem enviados ao armazenamento estável antes que os dados atualizados pela transação fossem gravados em disco.

17.10 Explique a finalidade do mecanismo de ponto de verificação. Com que freqüência os pontos de verificação devem ser realizados? Como a freqüência dos pontos de verificação afeta
   - O desempenho do sistema quando não ocorre uma falha?
   - O tempo gasto para a recuperação de uma falha do sistema?
   - O tempo gasto para a recuperação de uma falha do disco?

17.11 Explique como o gerenciador de buffer pode fazer com que o banco de dados se torne inconsistente se alguns registros de log pertencentes a um bloco não forem enviados ao armazenamento estável antes que o bloco seja enviado ao disco.

17.12 Explique os benefícios do logging lógico. Dê exemplos de uma situação em que o logging lógico é preferível ao logging físico e uma situação em que o logging físico é preferível ao logging lógico.

17.13 Explique a diferença entre uma falha do sistema e um "desastre".

17.14 Para cada um dos requisitos a seguir, identifique a melhor escolha de grau de durabilidade em um sistema de backup remoto:
   a. A perda de dados precisa ser evitada, mas alguma perda de disponibilidade pode ser tolerada.
   b. O commit da transação precisa ser realizado rapidamente, mesmo ao custo da perda de algumas transações confirmadas em um desastre.
   c. Um alto grau de disponibilidade e durabilidade é exigido, mas um tempo de execução maior para o protocolo de commit da transação é aceitável.

17.15 Um sistema de banco de dados Oracle utiliza registros de log de undo para oferecer uma visão de snapshot do banco de dados para transações somente leitura. A visão de snapshot reflete atualizações de todas as transações que foram confirmadas quando a transação somente leitura foi iniciada; as atualizações de todas as outras transações não são visíveis às transações somente leitura.

Descreva um esquema para o tratamento de buffer pelo qual as transações somente de leitura recebem uma visão de snapshot das páginas no buffer. Inclua detalhes de como usar o log para gerar a visão de snapshot, supondo que o algoritmo de recuperação avançado seja utilizado. Considere, por simplicidade, que uma operação lógica e seu undo afetem apenas uma única página.

Notas bibliográficas

Gray e Reuter [1993] é uma excelente fonte de informação em livro-texto sobre recuperação, incluindo interessantes detalhes de implementação e históricos. Bernstein *et al.* [1987] é uma antiga fonte de informação em livro-texto sobre controle de concorrência e recuperação.

Dois artigos antigos que apresentam o trabalho teórico inicial na área de recuperação são Davies [1973] e Bjork [1973]. Chandy *et al.* [1975], que descrevem os modelos analíticos para estratégias de rollback e recuperação em sistemas de banco de dados, é outro trabalho antigo nessa área.

Uma visão geral do esquema de recuperação do System R é apresentada por Gray *et al.* [1981]. Tutorial e artigos de levantamento sobre diversas técnicas de recuperação para sistemas de banco de dados incluem Gray [1978], Lindsay *et al.* [1980] e Verhofstad [1978]. Os conceitos de ponto de verificação difuso e dumps difusos são descritos em Lindsay *et al.* [1980]. Uma apresentação abrangente dos princípios de recuperação é oferecida por Haerder e Reuter [1983].

O que há de mais moderno em métodos de recuperação é ilustrado melhor pelo método de recuperação ARIES, descrito em Mohan *et al.* [1992] e Mohan [1990b]. ARIES e suas variantes são usadas em diversos produtos de banco de dados, incluindo IBM DB2 e Microsoft SQL Server. A recuperação no Oracle é descrita em Lahiri *et al.* [2001a].

As técnicas de recuperação especializadas para estruturas de índice são descritas em Mohan e Levine [1992] e Mohan [1993]; Mohan e Narang [1994] descrevem técnicas de recuperação para arquiteturas cliente-servidor, enquanto Mohan e Narang [1991] e Mohan e Narang [1992] descrevem técnicas de recuperação para arquiteturas de banco de dados paralelas.

O backup remoto para recuperação de desastre (perda de uma instalação de computação inteira, por exemplo, incêndio, inundação ou terremoto) é considerado em King *et al.* [1991] e Polyzois e Garcia-Molina [1994].

O Capítulo 25 lista referências pertencentes a transações de longa duração e questões de recuperação relacionadas.

PARTE 6

Mineração de dados e recuperação de informações

As consultas ao banco de dados normalmente são projetadas para extrair informações específicas, como o saldo de uma conta ou a soma dos saldos de conta de um cliente. Porém, as consultas projetadas para ajudar a formular uma estratégia corporativa normalmente exigem a agregação em uma escala muito maior e incluem análise estatística não expressa facilmente com os recursos da SQL que já vimos anteriormente. Essas consultas normalmente precisam acessar dados vindos de várias origens.

Um *depósito de dados* (ou *data warehouse*) é um repositório de dados reunidos de várias fontes e armazenados sob um esquema de banco de dados comum, unificado. Os dados armazenados no depósito são analisados por diversas agregações complexas e análises estatísticas. Além do mais, técnicas de descoberta de conhecimento podem ser utilizadas para tentar descobrir regras e padrões a partir dos dados. Por exemplo, um comerciante pode descobrir que certos produtos costumam ser comprados juntos, e pode usar essa informação para desenvolver suas estratégias de marketing. Esse processo de descoberta de conhecimento é chamado de *mineração de dados* (ou "data mining"). O Capítulo 18 trata dessas questões.

Em nossas discussões até o momento, focalizamos dados relativamente simples e bem estruturados. Contudo, existe uma enorme quantidade de dados textuais não estruturados na Internet, nas intranets dentro das organizações e nos computadores de usuários individuais. Os usuários querem encontrar informações relevantes a partir desse vasto corpo de informações principalmente textuais, usando mecanismos de consulta simples, como consultas de palavra-chave. O campo de recuperação de informações lida com a consulta de tais dados não estruturados e presta atenção particularmente à ordenação dos resultados da consulta. Embora essa área de pesquisa já tenha várias décadas, ela passou por um crescimento tremendo com o desenvolvimento da World Wide Web. O Capítulo 19 oferece uma introdução ao campo de recuperação de informações.

CAPÍTULO 18

Mineração e análise de dados

As empresas começaram a explorar os dados que surgem on-line para tomar melhores decisões sobre suas atividades, como quais itens estocar e como buscar melhor os clientes a fim de aumentar as vendas. Contudo, muitas de suas consultas são bem complicadas, e certos tipos de informação não podem ser extraídos mesmo usando SQL.

Várias técnicas e ferramentas estão disponíveis para ajudar com o suporte à decisão. Diversas ferramentas para análise de dados permitem que os analistas vejam os dados de diferentes maneiras. Outras ferramentas de análise calculam previamente resumos de quantidades de dados muito grandes, a fim de dar respostas rápidas às consultas. Os padrões SQL:1999 e SQL:2003 agora contêm construções adicionais para dar suporte à análise de dados. Outra técnica para adquirir conhecimento a partir dos dados é usar a *mineração de dados* (ou "data mining"), que visa detectar vários tipos de padrões em grandes volumes de dados. A mineração de dados suplementa diversos tipos de técnicas estatísticas com objetivos semelhantes.

Este capítulo aborda o apoio à decisão, incluindo o processamento analítico on-line, depósito de dados e mineração de dados.

Sistemas de apoio à decisão

De modo geral, as aplicações de banco de dados podem ser classificadas em sistemas de processamento de transação e apoio à decisão. Os sistemas de processamento de transação são sistemas que registram informações sobre transações, como informação de vendas de produtos para empresas, ou registro de curso e informação de notas para universidades. Os sistemas de processamento de transação são bastante utilizados hoje, e as organizações têm acumulado uma grande quantidade de informações geradas por esses sistemas. Os sistemas de apoio à decisão visam obter informações de alto nível a partir de informações detalhadas armazenadas nos sistemas de processamento de transação, além de usar as informações de alto nível para tomar uma série de decisões. Sistemas de apoio à decisão ajudam gerentes a decidir quais produtos estocar em uma loja, quais produtos fabricar em uma fábrica, ou quais dos candidatos devem ser admitidos em uma universidade.

Por exemplo, os bancos de dados da empresa normalmente contêm enormes quantidades de informação sobre clientes e transações. O tamanho do armazenamento de informações exigido pode variar até centenas de gigabytes, ou até mesmo terabytes, para grandes cadeias de varejo. As informações da transação para um varejista podem incluir o nome ou identificador (como número de cartão de crédito) do cliente, os itens comprados, o preço pago e as datas em que as compras foram feitas. As informações sobre os itens adquiridos podem incluir o nome do item, o fabricante, o número do modelo, a cor e o tamanho. As informações do cliente podem incluir histórico de crédito, receita anual, residência, idade e até mesmo o grau de instrução.

Esses grandes bancos de dados podem ser informações valiosas para a tomada de decisões de negócios, como quais itens estocar e quais descontos oferecer. Por exemplo, uma empresa de varejo pode observar um surto repentino nas vendas de camisas de flanela no Noroeste do Pacífico, pode observar que existe uma tendência e pode começar a estocar um número maior dessas camisas nas lojas dessa região. Como outro exemplo, uma empresa de carros pode descobrir, consultando seu banco de dados, que a maioria dos seus carros esportivos pequenos é comprada por jovens do sexo feminino cujas receitas anuais estão acima de $50.000. A

empresa pode, então, visar seu marketing para atrair mais dessas jovens para comprar seus carros esportivos pequenos, e pode evitar desperdiçar dinheiro tentando atrair outras categorias de pessoas para comprar esses carros. Nos dois casos, a empresa identificou padrões no comportamento do cliente e os utilizou para tomar decisões de negócios.

O armazenamento e a recuperação dos dados para apoio à decisão levantam várias questões:

- Embora muitas consultas de apoio à decisão possam ser escritas em SQL, outras não podem ser expressas em SQL ou não podem ser expressas com facilidade em SQL. Portanto, várias extensões da SQL foram propostas para facilitar a análise dos dados. A área de *processamento analítico on-line* (OLAP – On-Line Analytical Processing) lida com ferramentas e técnicas para análise de dados, que oferecem respostas quase instantâneas às consultas solicitando dados resumidos, embora o banco de dados possa ser extremamente grande. Na próxima seção, estudamos as extensões da SQL para análise de dados e as técnicas para o processamento analítico on-line.
- As linguagens de consulta de banco de dados não são adequadas ao desempenho de análises estatísticas detalhadas dos dados. Existem vários pacotes, como SAS e S++, que ajudam na análise estatística. Esses pacotes têm sido interligados aos bancos de dados, permitindo que grandes volumes de dados sejam armazenados no banco de dados e recuperados de forma eficiente para análise. O campo da análise estatística é uma disciplina grande por si só; para obter mais informações, consulte as referências nas notas bibliográficas.
- Grandes empresas possuem diversas origens de dados que precisam usar para tomar decisões de negócios. As fontes podem armazenar os dados sob diferentes esquemas. Por motivos de desempenho (bem como por motivos de controle da organização), as origens de dados normalmente não permitirão que outras partes da empresa apanhem dados por demanda.

  Para executar consultas de forma eficiente em dados tão diversificados, as empresas montaram *depósitos de dados* (data warehouses). Os depósitos de dados reúnem dados de várias origens sob um esquema unificado, em um único local. Assim, eles oferecem ao usuário uma única interface uniforme para os dados. Estudamos as questões de montagem e manutenção de depósitos de dados na seção "Depósito de dados".
- Técnicas de descoberta de conhecimento tentam descobrir automaticamente regras estatísticas e padrões a partir dos dados. O campo de *mineração de dados* combina as técnicas de descoberta de conhecimento inventadas pelos pesquisadores de inteligência artificial e analistas estatísticos, com técnicas de implementação eficientes, que lhes permitem ser usadas em bancos de dados extremamente grandes. A seção "Mineração de dados" discute a mineração de dados.

A área do **apoio à decisão** pode ser vista de forma geral como abordando todas essas áreas, embora algumas pessoas utilizem o termo em um sentido mais estrito, que exclui análise estatística e mineração de dados.

Análise de dados e OLAP

Embora a análise estatística complexa deva ser deixada para os pacotes estatísticos, os bancos de dados deverão admitir formas de análise de dados simples, comumente utilizadas. Como os dados armazenados nos bancos de dados normalmente são de grande volume, eles precisam ser resumidos em algum padrão se tivermos de derivar informações que os humanos podem usar.

Ferramentas OLAP admitem análise interativa de informações de resumo. Várias extensões da SQL foram desenvolvidas para dar suporte a ferramentas OLAP. Existem muitas tarefas, comumente utilizadas, que não podem ser feitas com as facilidades básicas da SQL para agregação e agrupamento. Alguns exemplos incluem encontrar percentis, ou distribuições cumulativas, ou agregados sobre janelas deslizantes nos dados ordenados seqüencialmente. Diversas extensões da SQL foram propostas recentemente para dar suporte a tais tarefas e foram implementadas em produtos como Oracle e IBM DB2.

Processamento analítico on-line

A análise estatística normalmente exige o agrupamento sobre vários atributos. Considere uma aplicação em que uma loja deseja descobrir quais tipos de roupas são populares. Vamos supor que as roupas sejam caracterizadas por seu *nome_item*, cor e tamanho, e que temos uma relação *vendas* com o esquema *vendas*(*nome_item, cor, tamanho, número*). Suponha que *nome_item* possa assumir os valores (skirt, dress, shirt, pant), *cor* possa assumir os valores (dark, pastel, white) e *tamanho* possa assumir os valores (small, medium, large).

Dada uma relação usada para análise de dados, podemos identificar alguns dos atributos como **atributos de medida**, pois medem algum valor e podem ser agrupados dessa forma. Por exemplo, o atributo *número* da relação *vendas* é um atributo de medida, pois mede o número de unidades vendidas. Alguns dos (ou todos os) outros atributos da relação são identificados como **atributos de dimensão**, pois definem as dimensões em que os atributos de medida, e resumos dos atributos de medida, são vistos. Na relação *vendas*,

tamanho: **all**

nome_item \ *cor*	dark	pastel	white	Total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

Figura 18.1 Tabulação cruzada de *vendas* por *nome_item* e *cor*.

nome_item, *cor* e *tamanho* são atributos de dimensão. (Uma versão mais realista da relação *vendas* teria dimensões adicionais, como tempo e local de vendas, e medidas adicionais, como valor monetário da venda.)

Os dados que podem ser modelados como atributos de dimensão e atributos de medida são chamados **dados multidimensionais**.

Para analisar os dados multidimensionais, um gerente pode querer ver os dados dispostos como aparecem na tabela da Figura 18.1. A tabela mostra números totais para diferentes combinações de *nome_item* e *cor*. O valor de *tamanho* é especificado como sendo **all**, indicando que os valores exibidos são um resumo de todos os valores de *tamanho*.

A tabela na Figura 18.1 é um exemplo de uma **tabulação cruzada** (ou ***cross-tab***, para resumir), também referenciada como uma **tabela pivô**. Em geral, uma tabulação cruzada é uma tabela em que os valores para um atributo (digamos, *A*) formam os cabeçalhos de linha, os valores para outro atributo (digamos, *B*) formam os cabeçalhos de coluna, e os valores em uma célula individual são derivados da seguinte forma. Cada célula pode ser identificada por (a_i, b_j), onde a_i é um valor para *A* e b_j um valor para *B*. Se houver no máximo uma tupla com qualquer valor (a_i, b_j), o valor na célula é derivado dessa única tupla (se houver); por exemplo, poderia ser o valor de um ou mais atributos da tupla. Se puder haver várias tuplas com um valor (a_i, b_j), o valor na célula precisa ser derivado pela agregação sobre as tuplas com esse valor. Em nosso exemplo, a agregação usada é a soma dos valores para o atributo *número*, por todos os valores para *tamanho*, conforme indicado por *tamanho*: **all** acima da tabulação cruzada na Figura 18.1. Em nosso exemplo, a tabulação cruzada também tem uma coluna extra e uma linha extra que armazena os totais das células na linha/coluna. A maioria das tabulações cruzadas possui essas linhas e colunas de resumo.

Uma tabulação cruzada é diferente das tabelas relacionais normalmente armazenadas nos bancos de dados, pois o número de colunas na tabulação cruzada depende dos dados reais. Uma mudança nos valores de dados pode resultar no acréscimo de mais colunas, o que não é desejável para armazenamento de dados. Porém, uma visão de tabulação cruzada é desejável para exibição aos usuários. É simples representar uma tabulação cruzada sem valores de resumo em uma forma relacional com um número fixo de colunas. Uma tabulação cruzada com linhas/colunas de resumo pode ser representada pela introdução de um valor especial **all** para representar subtotais, como na Figura 18.2. O padrão **SQL:1999**, na realidade, utiliza o valor **null** no lugar de **all**, mas, para evitar confusão com valores nulos regulares, continuaremos a usar **all**.

Considere as tuplas (skirt, **all**, **all**, 53) e (dress, **all**, **all**, 35). Obtemos essas tuplas eliminando as tuplas individuais com diferentes valores para *cor* e *tamanho*, e substituindo o valor de *número* por um agregado – a saber, sum. O valor **all** pode ser imaginado como representando o conjunto de todos os valores para um atributo. As tuplas com o valor **all** para as dimensões *cor* e *tamanho* podem ser obtidas por uma agregação sobre a relação *vendas* com um group by sobre a coluna *nome_item*. De modo semelhante, um group by sobre *cor*, *tamanho* pode ser usado para obter as tuplas com o valor **all** para *nome_item*, e um group by sem atributos (que simplesmente podem ser omitidos em SQL) pode ser usado para obter a tupla com o valor **all** para *nome_item*, *cor* e *tamanho*.

A generalização de uma tabulação cruzada, que é bidimensional, para *n* dimensões pode ser visualizada como um cubo multidimensional, chamado **cubo de dados**. A Figura 18.3 mostra um cubo de dados sobre a relação *vendas*. O cubo de dados possui três dimensões, a saber, *nome_item*, *cor* e *tamanho*, e o atributo de medida é *número*. Cada célula é identificada por valores para essas três dimensões. Cada célula no cubo de dados contém um valor, assim como na tabulação cruzada. Na Figura 18.3, o valor contido em uma célula aparece em uma das faces da célula; outras faces da célula aparecem em branco, se estiverem visíveis. Todas as células contêm valores, mesmo que não estejam visíveis.

nome_item	cor	tamanho	número
skirt	dark	all	8
skirt	pastel	all	35
skirt	white	all	10
skirt	all	all	53
dress	dark	all	20
dress	pastel	all	10
dress	white	all	5
dress	all	all	35
shirt	dark	all	14
shirt	pastel	all	7
shirt	white	all	28
shirt	all	all	49
pant	dark	all	20
pant	pastel	all	2
pant	white	all	5
pant	all	all	27
all	dark	all	62
all	pastel	all	54
all	white	all	48
all	all	all	164

Figura 18.2 Representação relacional dos dados da Figura 18.1.

O valor para uma dimensão pode ser **all**, quando a célula contém um resumo de todos os valores dessa dimensão, como no caso das tabulações cruzadas. O número de maneiras diferentes como as tuplas podem ser agrupadas para agregação pode ser grande. Na verdade, para uma tabela com n dimensões, a agregação pode ser realizada com o agrupamento sobre cada um dos 2^n subconjuntos das n dimensões.[1]

Um sistema de processamento analítico on-line, ou OLAP, é um sistema interativo que permite que um analista veja diferentes resumos dos dados multidimensionais. A palavra *on-line* indica que um analista precisa ser capaz de solicitar novos resumos e obter respostas on-line, dentro de alguns segundos, e não deve ser forçado a esperar muito tempo para ver o resultado de uma consulta.

Com um sistema OLAP, um analista de dados pode examinar diferentes tabulações cruzadas sobre os mesmos dados, selecionando interativamente os atributos na tabulação cruzada. Cada tabulação cruzada é uma visão bidimensional em um cubo de dados multidimensional. Por exemplo, o analista pode selecionar uma tabulação cruzada sobre *nome_item* e *tamanho* ou uma tabulação cruzada sobre *cor* e *tamanho*. A operação de mudar as dimensões usadas em uma tabulação cruzada é denominada **pivotar**.

Um sistema OLAP também oferece outra funcionalidade. Por exemplo, o analista pode querer ver uma tabulação cruzada sobre *nome_item* e *cor* para um valor fixo de *tamanho*, por exemplo, large, em vez da soma por todos os tamanhos. Tal operação é considerada como **slicing**, pois pode ser imaginada como vendo uma fatia (*slice*) do cubo de dados. A operação às vezes é chamada de **dicing**, principalmente quando os valores para as múltiplas dimensões são fixos.

Quando uma tabulação cruzada é usada para ver um cubo multidimensional, os valores dos atributos de dimensão que não fazem parte da tabulação cruzada aparecem acima da tabulação cruzada. O valor de tal atributo pode ser **all**, como mostra a Figura 18.1, indicando que os dados na tabulação cruzada são um resumo sobre todos os valores para o atributo. Slicing/dicing simplesmente consiste em selecionar valores específicos para esses atributos, que são então exibidos em cima da tabulação cruzada.

Sistemas OLAP permitem que os usuários vejam dados em qualquer nível de granularidade desejado. A operação de passar de dados com granularidade mais detalhada para uma granularidade mais esparsa (por meio da agregação) é chamada de **subir**. Em nosso exemplo, começando do cubo de dados na tabela *vendas*, obtemos nossa tabulação cruzada de exemplo subindo pelo atributo *tamanho*. A operação oposta – de mover de dados com granularidade mais esparsa para dados com granularidade mais detalhada – é chamada de **descer**. Nitidamente, os dados com granularidade

1. O agrupamento sobre o conjunto de todas as n dimensões só é útil se a tabela puder ter duplicatas.

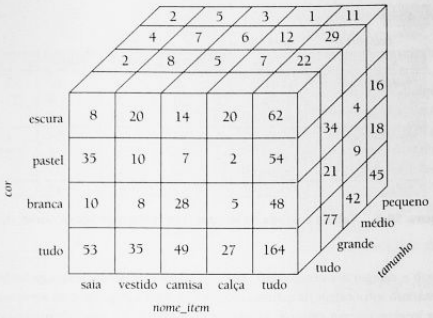
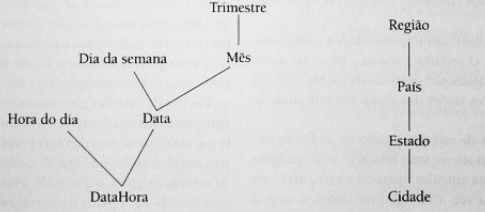


Figura 18.3 Cubo de dados tridimensional.

mais detalhada não podem ser gerados a partir de dados com granularidade mais esparsa; eles precisam ser gerados pelos dados originais ou por dados de resumo com granularidade ainda mais detalhada.

Os analistas podem querer ver uma dimensão em diferentes níveis de detalhe. Por exemplo, um atributo do tipo **datahora** contém uma data e uma hora do dia. Usar um tempo exato até um segundo (ou menos) pode não ser significativo: um analista que está interessado na hora bruta do dia pode examinar apenas o valor da hora. Um analista interessado nas vendas por dia da semana pode mapear a data para um dia da semana e examinar apenas isso. Outro analista pode estar interessado em agregações durante um mês, ou trimestre, ou para um ano inteiro.

Os diferentes níveis de detalhe para um atributo podem ser organizados em uma **hierarquia**. A Figura 18.4a mostra uma hierarquia sobre o atributo **datetime**. Como outro exemplo, a Figura 18.4b mostra uma hierarquia sobre local, com a cidade estando no final da hierarquia, estado acima dela, país no próximo nível e região estando no nível superior. Em nosso exemplo anterior, as roupas podem ser agrupadas por categoria (por exemplo, menswear ou womenswear); *categoria*, então, estaria acima de *nome_item* em nossa hierarquia sobre roupas. No nível de valores reais,



(a) Hierarquia de tempo (b) Hierarquia de local

Figura 18.4 Hierarquias sobre dimensões.

tamanho: **all**

categoria	*nome_item*	*cor*				
		dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pant	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Figura 18.5 Tabela cruzada de *vendas* com hierarquia sobre *nome_item*.

saias e vestidos cairiam sob a categoria womenswear, enquanto calças e camisas estariam sob a categoria menswear.

Um analista pode estar interessado em exibir as vendas de roupas divididas como menswear e womenswear, e não nos valores individuais. Depois de exibir os agregados no nível de womenswear e menswear, um analista pode *descer na hierarquia* para examinar os valores individuais. Um analista examinando no nível detalhado por *subir na hierarquia* e examinar as agregações no nível mais esparso. Os dois níveis podem ser exibidos sobre a mesma tabulação cruzada, como na Figura 18.5.

Implementação do OLAP

Os sistemas OLAP mais antigos usavam arrays multidimensionais na memória para armazenar os cubos de dados, e são conhecidos como sistemas **OLAP multidimensional** (**MOLAP**). Mais tarde, facilidades OLAP foram integradas aos sistemas relacionais, com os dados armazenados em um banco de dados relacional. Esses sistemas são conhecidos como sistemas **OLAP relacional** (**ROLAP**). Os sistemas híbridos, que armazenam alguns resumos na memória e armazenam a base de dados e outros resumos em um banco de dados relacional, são chamados sistemas **OLAP híbrido** (**HOLAP**).

Muitos sistemas OLAP são implementados como sistemas cliente-servidor. O servidor contém o banco de dados relacional e também quaisquer cubos de dados MOLAP. Os sistemas cliente obtêm visões dos dados comunicando-se com o servidor.

Um modo simples de calcular o cubo de dados inteiro (todos os agrupamentos) em uma relação é usar qualquer algoritmo padrão para calcular operações agregadas, um agrupamento de cada vez. Um algoritmo simples exigiria uma grande quantidade de varreduras da relação. Uma otimização simples é calcular uma agregação sobre, digamos, (*nome_item*, cor) a partir de uma agregação (*nome_item*, *cor*, *tamanho*), em vez da relação original.

Para as funções de agregação SQL padrão, podemos calcular uma agregação com agrupamento sobre um conjunto de atributos A a partir de uma agregação com agrupamento sobre um conjunto de atributos B se $A \subseteq B$; você pode fazer isso como um exercício (ver Exercício 18.8), mas observe que, para calcular **avg**, também precisamos do valor de **count**. (Para algumas funções de agregação fora do padrão, como median, as agregações não podem ser calculadas dessa forma; a otimização descrita aqui não se aplica a cada uma das funções agregadas *não decompostas*.) A quantidade de dados lidos cai bastante calculando uma agregação a partir de outra agregação, no lugar da relação original. Outras melhorias são possíveis; por exemplo, vários agrupamentos podem ser calculados sobre uma única varredura dos dados. Veja, nas notas bibliográficas, referências a algoritmos para calcular cubos de dados com eficiência.

As antigas implementações do OLAP calculavam previamente e armazenavam cubos de dados inteiros, ou seja, agrupamentos sobre todos os subconjuntos dos atributos de dimensão. O cálculo prévio permite que as consultas OLAP sejam respondidas dentro de alguns segundos, até mesmo em datasets que podem conter milhões de tuplas, chegando a gigabytes de dados. Porém, existem 2^n agrupamentos com n atributos de dimensão; as hierarquias sobre atributos aumentam o número ainda mais. Como resultado, o cubo de dados inteiro normalmente é maior do que a relação original que formou o cubo de dados e, em muitos casos, não é viável armazenar o cubo de dados inteiro.

Em vez de calcular previamente e armazenar todos os agrupamentos possíveis, faz sentido calcular previamente e armazenar alguns dos agrupamentos, calculando outros por demanda. No lugar de calcular consultas a partir da relação original, o que pode levar muito tempo, podemos calculá-las a partir de outras consultas previamente calculadas. por exemplo, suponha que uma consulta exija resumos por (*nome_item*, *cor*), que não foi previamente calculado. O resultado da consulta pode ser calculado a partir dos resumos por (*nome_item*, *cor*, *tamanho*), se

isso tiver sido previamente calculado. Veja, nas notas bibliográficas, referências sobre como selecionar um bom conjunto de agrupamentos para cálculo prévio, dados limites no armazenamento disponível para resultados previamente calculados.

Os dados em um cubo de dados não podem ser gerados por uma única consulta SQL, usando as construções **group by** básicas, pois as agregações são calculadas para vários agrupamentos diferentes dos atributos de dimensão. A próxima seção discute as extensões SQL para dar suporte à funcionalidade OLAP.

Agregação estendida

A funcionalidade da agregação SQL-92 é limitada, de modo que várias extensões foram implementadas por diferentes bancos de dados. Porém, o padrão SQL:1999 define um rico conjunto de funções de agregação, que resumimos nesta seção e nas duas seguintes. Os bancos de dados Oracle e IBM DB2 admitem a maior parte desses recursos, e, sem dúvida, outros bancos de dados admitirão esses recursos no futuro próximo.

As novas funções de agregação sobre atributos isolados são desvio-padrão e variância (**stddev** e **variance**). O desvio-padrão é a raiz quadrada da variância.[2] Alguns sistemas de banco de dados admitem outras funções de agregação, como mediana e moda. Alguns sistemas de banco de dados até mesmo permitem que os usuários acrescentem novas funções de agregação.

A SQL:1999 também admite uma nova classe de **funções de agregação binárias**, que podem calcular resultados estatísticos sobre pares de atributos; elas incluem correlações, covariâncias e curvas de regressão, que oferecem uma linha aproximando a relação entre os valores do par de atributos. As definições dessas funções podem ser encontradas em qualquer livro-texto padrão sobre estatística, como aqueles listados nas notas bibliográficas.

SQL:1999 também admite generalizações da construção **group by**, usando as construções **cube** e **rollup**. Um uso representativo da construção **cube** é:

```
select nome_item, cor, tamanho, sum(número)
from   vendas
group by cube(nome_item, cor, tamanho)
```

Essa consulta calcula a união de oito agrupamentos diferentes da relação *vendas*:

```
{ (nome_item, cor, tamanho), (nome_item, cor),
  (nome_item, tamanho),
  (cor, tamanho), (nome_item), (cor), (tamanho), ( ) }
```

onde ( ) indica uma lista **group by** vazia.

Para cada agrupamento, o resultado contém o valor nulo para atributos ausentes no agrupamento. Por exemplo, a tabela na Figura 18.2, com ocorrências de **all** substituídas por *null*, pode ser calculada pela consulta

```
select nome_item, cor, sum(número)
from   vendas
group by cube(nome_item, cor)
```

Uma construção **rollup** representativa é

```
select nome_item, cor, tamanho, sum(número)
from   vendas
group by rollup(nome_item, cor, tamanho)
```

Aqui, somente quatro agrupamentos são gerados:

```
{ (nome_item, cor, tamanho), (nome_item, cor),
  (nome_item), ( ) }
```

O rollup pode ser usado para gerar agregações em vários níveis de uma hierarquia sobre uma coluna. Por exemplo, suponha que tenhamos uma tabela *itemcategoria*(*nome_item*, categoria) dando a categoria de cada item. Depois a consulta

```
select categoria, nome_item, sum(número)
from   vendas, itemcategoria
where  vendas.nome_item = itemcategoria.nome_item
group by rollup(categoria, nome_item)
```

daria um resumo hierárquico por *nome_item* e por *categoria*.

Vários **rollups** e **cubos** podem ser usados em um único grupo por cláusula. Por exemplo, a consulta a seguir

```
select nome_item, cor, tamanho, sum(número)
from   vendas
group by rollup(nome_item), rollup(cor, tamanho)
```

gera os agrupamentos

```
{ (nome_item, cor, tamanho), (nome_item, cor),
  (nome_item),
  (cor, tamanho), (cor), ( ) }
```

Para entender por que, observe que **rollup**(*nome_item*) gera dois agrupamentos, {(*nome_item*), ( )}, e **rollup**(*cor*, *tama-*

2. O padrão SQL:1999 na realidade admite dois tipos de variância, chamados *variância de população* e *variância de amostra*; da mesma forma, existem dois tipos de desvio-padrão. As definições dos dois tipos diferem ligeiramente; veja os detalhes em um livro-texto sobre estatística.

nho) gera três agrupamentos, {(*cor*, *tamanho*), (*cor*), ( )}. O produto cruzado dos dois nos oferece os seis agrupamentos mostrados.

Conforme mencionamos na seção "Processamento analítico on-line", a SQL:1999 usa o valor **null** para indicar o sentido normal de nulo e também **all**. Esse uso dual de **null** pode causar ambigüidade se os atributos usados em uma cláusula rollup ou cube tiver valores nulos. A função **grouping** pode ser aplicada sobre um atributo; ela retorna 1 se o valor for um valor nulo representando **all**, e retorna 0 em todos os outros casos. Considere a seguinte consulta:

```
select nome_item, cor, tamanho, sum(número),
    grouping(nome_item) as nome_item-flag,
    grouping(cor) as cor-flag,
    grouping(tamanho) as tamanho-flag
from vendas
group by cube(nome_item, cor, tamanho)
```

A saída é igual à da versão da consulta sem **grouping**, mas com três colunas extras, chamadas flagnomeitem, flagcor e flagtamanho. Em cada tupla, o valor de um campo de flag é 1 se o campo correspondente for um nulo representando **all**.

Em vez de usar tags para indicar nulos que representam **all**, podemos substituir o valor nulo por um valor à nossa escolha.

```
decode(grouping(nome_item), 1, 'all', nome_item)
```

Essa expressão retorna o valor "all" se o valor de *nome_item* for um nulo correspondente a **all**, e retorna o valor real de *nome_item* em caso contrário. A expressão pode ser usada no lugar de *nome_item* na cláusula select para obter "tudo" na saída da consulta, no lugar de nulos representando **all**.

Nem o **rollup** nem a cláusula **cube** dão controle completo dos agrupamentos que são gerados. Por exemplo, não podemos usá-los para especificar que queremos apenas agrupamentos {(*cor*, *tamanho*), (*tamanho*, *nome_item*)}. Esses agrupamentos restritos podem ser gerados usando a construção **grouping** na cláusula **having**; deixamos os detalhes como um exercício para você.

Ordenação

Encontrar a posição de um valor em um conjunto maior é uma operação comum. Por exemplo, podemos querer atribuir aos alunos uma classificação na sua turma com base em suas notas totais, com a classificação 1 indo para o aluno com as maiores notas, a classificação 2 para o aluno com as próximas maiores notas, e assim por diante. Embora tais consultas possam ser expressas na SQL-92, elas são difíceis de expressar e ineficazes de avaliar. Os programadores normalmente lançam mão de escrever a consulta parcialmente na SQL e parcialmente em uma linguagem de programação. Um tipo de consulta relacionado é localizar o percentil em que um valor em um (multi)conjunto pertence, por exemplo, a terça parte interior, a terça parte do meio ou a terça parte de cima. Estudamos aqui o suporte da SQL:1999 para esses tipos de consultas.

A classificação é feita em conjunto com uma especificação **order by**. Suponha que recebamos uma relação *notas_aluno*(*id_aluno*, *notas*), que armazena as notas obtidas por aluno. A consulta a seguir oferece a classificação de cada aluno.

```
select id_aluno, rank( ) over (order by (notas) desc)
as ordem_a
from notas_aluno
```

Observe que a ordem das tuplas na saída não é definida, de modo que elas podem ser ordenadas por rank. Uma cláusula **order by** é necessária para que fiquem em ordem, como vemos a seguir.

```
select id_aluno, rank ( ) over (order by (notas) desc)
as ordem_a
from notas_aluno order by ordem_a
```

Uma questão básica com a classificação é como lidar com o caso de várias tuplas que são iguais no(s) atributo(s) de ordenação. Em nosso exemplo, isso significa decidir o que fazer se houver dois alunos com as mesmas notas. A função **rank** dá a mesma classificação a todas as tuplas que são iguais nos atributos de **order by**. Por exemplo, se a nota mais alta for compartilhada por dois alunos, os dois teriam a classificação 1. A próxima classificação dada seria 3, e não 2, de modo que se três alunos receberem a próxima nota mais alta, todos eles receberiam a classificação 3, e o(s) próximo(s) aluno(s) receberia(m) a classificação 5, e assim por diante. Há também uma função **dense_rank**, que não cria lacunas na ordenação. No exemplo anterior, as tuplas com o segundo valor mais alto receberiam a classificação 2, e as tuplas com o terceiro valor mais alto receberiam a classificação 3, e assim por diante.

A classificação pode ser feita dentro das partições dos dados. Por exemplo, suponha que tenhamos uma relação adicional *seção_aluno*(*id_aluno*, *seção*) que armazena, para cada aluno, a seção em que o aluno estuda. A consulta a seguir, então, dá a classificação dos alunos dentro de cada seção.

```
select id_aluno, seção,
    rank ( ) over (partition by seção order by notas
desc) as ordem_sec
from notas_aluno, seção_aluno
where notas_aluno.id_aluno = seção_aluno.id_aluno
order by seção, ordem_sec
```

A cláusula **order by** mais externa resulta em tuplas por seção, e dentro de cada seção, pela classificação.

Várias expressões **rank** podem ser usadas dentro de uma única instrução select; assim, podemos obter a classificação geral e a classificação dentro da seção, usando duas expressões **rank** na mesma cláusula **select**. Uma questão interessante é o que acontece quando a classificação (possivelmente com particionamento) ocorre junto com uma cláusula **group by**. Nesse caso, a cláusula **group by** é aplicada primeiro, e o particionamento e a classificação são feitos sobre os resultados do group by. Assim, os valores agregados podem então ser usados para a classificação. Por exemplo, suponha que tenhamos notas para cada aluno para cada uma de várias matérias. Para classificar os alunos pela soma de suas notas em diferentes matérias, podemos usar uma cláusula **group by** para calcular as notas agregadas para cada aluno e depois classificar os alunos pela soma agregada. Deixamos os detalhes como um exercício para você.

As funções de classificação podem ser usadas para encontrar as n tuplas superiores embutindo uma consulta de classificação dentro de uma consulta de nível mais externo; deixamos os detalhes como um exercício. Observe que as n inferiores são simplesmente as mesmas que as n superiores com uma ordem de classificação reversa. Vários sistemas de banco de dados oferecem extensões da SQL não padronizadas para especificar diretamente que somente os n resultados superiores são exigidos; essas extensões não exigem a função rank e simplificam a tarefa do otimizador, mas (atualmente) não são tão genéricas, pois não admitem particionamento.

A SQL:1999 também especifica várias outras funções que podem ser usadas no lugar de **rank**. Por exemplo, **percent_rank** de uma tupla indica a classificação da tupla como uma fração. Se houver n tuplas na partição[3] e a classificação da tupla for r, então sua classificação percentual é definida como $(r-1)/(n-1)$ (e como nula, se houver apenas uma tupla na partição). A função **cume_dist**, abreviação de "cumulative distribution" (distribuição cumulativa), para uma tupla, é definida como p/n, onde p é o número de tuplas na partição com valores de ordenação precedendo ou iguais ao valor de ordenação da tupla e n é o número de tuplas na partição. A função **row_number** classifica as linhas e dá a cada linha um número exclusivo correspondente à sua posição na ordem de classificação; linhas diferentes com o mesmo valor de classificação receberiam diferentes números de linha, em um padrão não determinista.

Finalmente, para determinada constante n, a função de classificação **ntile**(n) apanha as tuplas em cada partição na ordem especificada e as divide em n buckets com o mesmo número de tuplas.[4] Para cada tupla, **ntile**(n) então indica o número do bucket em que está colocada, com números de bucket começando com 1. Essa função é particularmente útil para construir histogramas com base em percentis. Por exemplo, podemos classificar funcionários por salário e usar **ntile**(3) para encontrar em qual intervalo (terça parte inferior, do meio ou superior) cada funcionário está, e calcular o salário total ganho pelos funcionários em cada intervalo:

```
select threetile, sum(salário)
from (
    select salário, ntile(3) over (order by (salário)) as
threetile
    from funcionário) as s
group by threetile
```

A presença de valores nulos pode complicar a definição da classificação, pois não fica claro onde eles devem ocorrer primeiro na ordem de classificação. A SQL:1999 permite que o usuário especifique onde eles devem ocorrer usando **nulls first** ou **nulls last**, por exemplo:

```
select id_aluno, rank ( ) over (order by notas desc
nulls last) as ordem_a
from notas_aluno
```

Janelas

Um exemplo de uma consulta em *janela* é uma consulta que, dados os valores de venda para cada data, calcula para cada data a média das vendas naquele dia, no dia anterior e no dia seguinte; essas consultas de média móvel são usadas para suavizar variações aleatórias. Outro exemplo de uma consulta em janela é aquela que encontra o saldo cumulativo em uma conta, dada uma relação especificando os depósitos e retiradas em uma conta. Essas consultas são difíceis ou impossíveis (dependendo da consulta exata) de expressar na SQL básica.

A SQL:1999 oferece um recurso de janela para dar suporte a tais consultas. Ao contrário do **group by**, a mesma tupla pode existir em várias janelas. Suponha que tenhamos recebido uma relação *transação*(*número_conta, data_hora, valor*), onde *valor* é positivo para um depósito e negativo para uma retirada. Consideramos que existe no máximo uma transação por valor de *data_hora*.

Considere a consulta

3. O conjunto inteiro é tratado como uma única partição se nenhuma partição explícita for usada.

4. Se o número total de tuplas em uma partição não for divisível por n, então o número de tuplas em cada bucket pode diferir em no máximo 1. As tuplas com o mesmo valor para o atributo de ordenação podem ser atribuídas a diferentes buckets, de forma não determinista, a fim de igualar o número de tuplas em cada bucket.

```
select numero_conta, data_hora,
    sum(valor) over
        (partition by numero_conta
        order by data_hora
        rows unbounded preceding)
    as saldo
from transação
order by numero_conta, data_hora
```

Ela gera os saldos cumulativos sobre cada conta imediatamente antes de cada transação sobre a conta; o saldo cumulativo da conta é a soma de valores de todas as transações anteriores na conta.

A cláusula **partition by** particiona tuplas por número de conta, de modo que, para cada linha, somente as tuplas em sua partição sejam consideradas. Uma janela é criada para cada tupla; as palavras-chave **rows unbounded preceding** especificam que a janela para cada tupla consiste em todas as tuplas na partição que a precedem na ordem especificada (aqui, em ordem crescente de *data_hora*). A função agregada **sum**(*valor*) é aplicada a todas as tuplas na janela. Observe que a consulta não usa uma cláusula **group by**, pois existe uma tupla de saída para cada tupla na relação *transação*.

Embora a consulta possa ser escrita sem essas construções estendidas, ela seria um tanto difícil de formular. Observe também que diferentes janelas podem se sobrepor; ou seja, uma tupla pode estar presente em mais de uma janela.

Outros tipos de janelas podem ser especificados. Por exemplo, para obter uma janela contendo as 10 linhas anteriores para cada linha, podemos especificar **rows** 10 **preceding**. Para obter uma janela contendo as linhas atual, anterior e seguinte, podemos usar **between rows** 1 **preceding and** 1 **following**. Para obter as linhas anteriores e a linha atual, podemos dizer **between rows unbounded preceding and current**. Observe que, se a ordenação for sobre um atributo não-chave, o resultado não é determinista, pois a ordem de tuplas não é totalmente definida.

Podemos até mesmo especificar janelas por intervalos de valores, em vez do número de linhas. Por exemplo, suponha que o valor de ordenação de uma tupla seja *v*; então, **range between** 10 **preceding and current row** daria tuplas cujo valor de ordenação é entre *v* – 10 e *v* (ambos os valores inclusive). Ao lidar com datas, podemos usar **range interval** 10 **day preceding** para obter uma janela contendo tuplas dentro dos 10 dias anteriores, mas não incluindo a data da tupla.

Claramente, a funcionalidade de janelas da SQL:1999 é muito rica e pode ser usada para escrever consultas bem complexas, com uma pequena quantidade de esforço.

Depósito de dados

Grandes empresas têm presenças em muitos lugares, cada uma delas podendo gerar um grande volume de dados. Por exemplo, grandes cadeias de revenda possuem centenas ou milhares de lojas, enquanto companhias de seguro podem ter dados de milhares de filiais locais. Além do mais, grandes organizações possuem uma estrutura organizacional interna bastante complexa, e por isso diferentes dados podem estar presentes em diferentes locais, ou em diferentes sistemas operacionais, ou sob diferentes esquemas. Por exemplo, os dados de problema de manufatura e os dados de reclamação do cliente podem ser armazenados em diferentes sistemas de banco de dados. Os que tomam decisões corporativas exigem o acesso a informações de todas essas origens. A preparação de consultas em origens individuais é desajeitada e ineficaz. Além do mais, as origens de dados só podem armazenar dados atuais, enquanto os que tomam decisões podem precisar de acesso também a dados do passado; por exemplo, informações sobre como os padrões de compra mudaram no ano passado poderiam ser de grande importância. Depósitos de dados oferecem uma solução para esses problemas.

Um **depósito de dados** (ou "data warehouse") é um repositório (ou arquivamento) de informações colhidas de várias origens, armazenadas sob um esquema unificado, em um único local. Uma vez reunidos, os dados são armazenados por muito tempo, permitindo o acesso a dados históricos. Assim, os depósitos de dados oferecem ao usuário uma única interface consolidada para os dados, facilitando a escrita de consultas de apoio à decisão. Além do mais, acessando informações para apoio à decisão a partir de um depósito de dados, quem toma decisões pode garantir que os sistemas de processamento de transação on-line não serão afetados pela carga de trabalho de apoio à decisão.

Componentes de um depósito de dados

A Figura 18.6 mostra a arquitetura de um depósito de dados típico, e ilustra a coleta de dados, o armazenamento de dados e o suporte da consulta e análise de dados. Entre as questões a serem enfrentadas na montagem de um depósito de dados estão as seguintes:

- Quando e como coletar dados. Em uma **arquitetura controlada pela origem** para a coleta de dados, as origens de dados transmitem novas informações, seja continuamente (quando ocorre o processamento da transação) ou periodicamente (à noite, por exemplo). Em uma **arquitetura controlada por destino**, o depósito de dados envia periodicamente solicitações para novos dados às origens.

  A menos que as atualizações nas origens sejam replicadas no depósito por meio do commit em duas fases, o depósito nunca estará muito atualizado com as origens. O commit em duas fases normalmente é muito dispen-

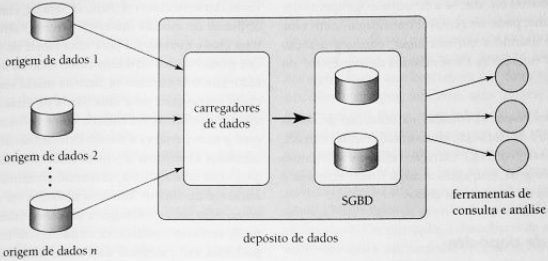


Figura 18.6 Arquitetura de depósito de dados.

dioso para ser uma opção, de modo que os depósitos de dados normalmente possuem dados ligeiramente desatualizados. Isso, porém, normalmente não é um problema para os sistemas de apoio à decisão.

- **Que esquema utilizar.** As origens de dados que foram construídas de forma independente provavelmente terão diferentes esquemas. Na verdade, elas podem até mesmo usar diferentes modelos de dados. Parte da tarefa de um depósito é realizar a integração de esquema e converter dados para o esquema integrado antes que eles sejam armazenados. Como resultado disso, os dados armazenados no depósito não são apenas uma cópia dos dados nas origens. Em vez disso, eles podem ser imaginados como uma visão materializada dos dados nas origens.
- **Transformação e limpeza de dados.** A tarefa de corrigir e pré-processar dados é chamada de **limpeza de dados**. As origens de dados normalmente entregam dados com diversas inconsistências menores, que podem ser corrigidas. Por exemplo, os nomes normalmente possuem erros de digitação, e os endereços podem ter erros em campos de rua/bairro/cidade, ou códigos postais informados incorretamente. Estes podem ser corrigidos até certo ponto consultando-se um banco de dados de nomes de rua e códigos postais em cada cidade. A combinação aproximada de dados exigidos para essa tarefa é considerada como **pesquisa difusa**.

  As listas de endereço coletadas de várias origens podem ter duplicatas que precisam ser eliminadas em uma **operação merge-purge** (essa operação também é conhecida como **deduplicação**). Os registros para os vários indivíduos em uma casa podem ser agrupados de modo que apenas uma correspondência seja enviada para cada casa; essa operação é chamada de **householding**.

  Os dados podem ser **transformados** de maneiras diferentes da limpeza, como a mudança de unidades de medida, ou a conversão dos dados para um esquema diferente, pela junção de dados de várias relações de origem. Os depósitos de dados normalmente possuem ferramentas gráficas para dar suporte à transformação de dados. Essas ferramentas permitem que a transformação seja especificada como caixas, e linhas podem ser criadas entre as caixas para indicar o fluxo dos dados. As caixas condicionais podem rotear dados para um próximo passo apropriado na transformação. Veja, na Figura 29.7, um exemplo de uma transformação especificada por meio da ferramenta gráfica fornecida pelo Microsoft SQLServer.
- **Como propagar atualizações.** As atualizações sobre as relações nas origens de dados precisam ser propagadas para o depósito de dados. Se as relações no depósito de dados forem exatamente as mesmas daquelas na origem de dados, a propagação será direta. Se não forem, o problema de propagação de atualizações é basicamente o problema de *manutenção de visão*, que foi discutido na seção "Views materializadas" do Capítulo 14.
- **Quais dados resumir.** Os dados brutos gerados por um sistema de processamento de transação podem ser muito grandes para serem armazenados on-line. Porém, podemos responder a muitas consultas mantendo apenas dados de resumo obtidos pela agregação em uma relação, em vez de manter a relação inteira. Por exemplo, em vez de armazenar dados sobre cada venda de roupas possível, podemos armazenar o total de vendas de roupas por *nome_item* e categoria.

  Suponha que uma relação r tenha sido substituída por uma relação de resumo s. Os usuários ainda podem ter permissão para fazer consultas como se a relação r es-

tivesse disponível on-line. Se a consulta exige apenas dados de resumo, pode ser possível transformá-la em uma equivalente usando s em seu lugar; consulte a seção "Estimando estatísticas de resultados de expressão" do Capítulo 14.

As diferentes etapas envolvidas na obtenção de dados para um depósito de dados são chamadas de tarefas **extract, transforme e load** (ou **ETL**): extração referente à obtenção de dados das origens, enquanto a carga (load) refere-se à carga dos dados no depósito de dados.

Esquemas de depósito

Os depósitos de dados normalmente possuem esquemas que são projetados para análise de dados, usando ferramentas do tipo OLAP. Assim, os dados normalmente são multidimensionais, com atributos de dimensão e atributos de medida. As tabelas contendo dados multidimensionais são denominadas **tabelas de fatos**, e normalmente são muito grandes. Uma tabela registrando informações de vendas para um comércio varejista, com uma tupla para cada item que é vendido, é um exemplo típico de uma tabela de fatos. As dimensões da tabela *vendas* incluiriam o que o item é (normalmente, um identificador de item, como aquele usado nos códigos de barra), a data em que o item foi vendido, de qual local (loja) o item foi vendido, qual cliente comprou o item e assim por diante. Os atributos de medida podem incluir o número de itens vendidos e o preço dos itens.

Para reduzir os requisitos de armazenamento, os atributos de dimensão normalmente são identificadores curtos, que são chaves estrangeiras para outras tabelas, chamadas **tabelas de dimensão**. Por exemplo, uma tabela de fatos *vendas* teria atributos *id_item*, *id_loja*, *id_cliente* e *data*, e os atributos de medida *número* e *preço*. O atributo *id_loja* é uma chave estrangeira para uma tabela de dimensão *loja*, que possui outros atributos como local da loja (cidade, estado, país). O atributo *id_item* da tabela *vendas* seria uma chave estrangeira para uma tabela de dimensão *info_item*, que teria informações como o nome do item, a categoria à qual o item pertence e outros detalhes do item, como cor e tamanho. O atributo *id_cliente* seria uma chave estrangeira para uma tabela *cliente*, contendo atributos como nome e endereço do cliente. Também podemos ver o atributo *data* como uma chave estrangeira para uma tabela *cliente*, contendo atributos como nome e endereço do cliente. Também podemos ver o atributo *data* como uma chave estrangeira para uma tabela *info_data*, dando o mês, o trimestre e o ano de cada data.

O esquema resultante aparece na Figura 18.7. Tal esquema, com uma tabela de fatos, várias tabelas de dimensão e chaves estrangeiras da tabela de fatos para as tabelas de dimensão, é chamado **esquema de estrela**. Projetos de depósito de dados mais complexos podem ter vários níveis de tabelas de dimensão; por exemplo, a tabela *info_item* pode ter um atributo *id_fabricante*, que é uma chave estrangeira para outra tabela, dando detalhes do fabricante. Esses esquemas são chamados **esquemas floco de neve**. Projetos complexos de depósito de dados também podem ter mais de uma tabela de fatos.

Mineração de dados

O termo **mineração de dados** (ou "data mining") refere-se, em geral, ao processo de analisar grandes bancos de dados de forma semi-automática para encontrar padrões úteis. Assim

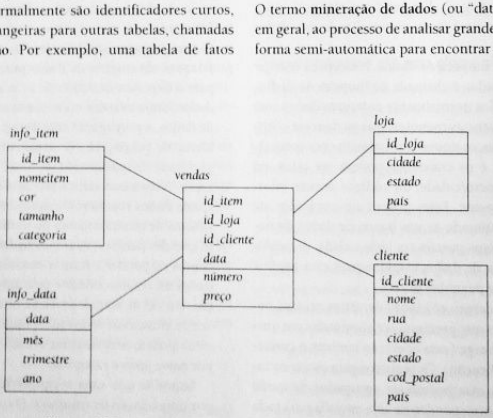


Figura 18.7 Esquema de estrela para um depósito de dados.

como a descoberta de conhecimento na inteligência artificial (também chamada aprendizado de máquina) ou na análise estatística, a mineração de dados tenta descobrir regras e padrões a partir dos dados. Porém, a mineração de dados difere do aprendizado de máquina e da estatística porque lida com grandes volumes de dados, armazenados principalmente no disco. Ou seja, a mineração de dados lida com "descoberta de conhecimento nos bancos de dados".

Alguns tipos de conhecimento descoberto de um banco de dados podem ser representados por um conjunto de regras. A seguir está um exemplo de uma regra, citada informalmente: "Jovens mulheres com renda anual maior que $50.000 são as pessoas mais prováveis de comprar carros esportivos". Naturalmente, essas regras não são universalmente verdadeiras, e possuem graus de "suporte" e "confiança", conforme veremos. Outros tipos de conhecimento são representados por equações relacionando diferentes variáveis entre si, ou por outros mecanismos para prever resultados quando os valores de algumas variáveis são conhecidos.

Existem diversos tipos possíveis de padrões que podem ser úteis, e diferentes técnicas são utilizadas para encontrar diferentes tipos de padrões. Estudaremos alguns exemplos de padrões e veremos como eles podem ser derivados automaticamente por um banco de dados.

Normalmente, existe um componente manual para a mineração de dados, consistindo no pré-processamento dos dados para um formato aceitável aos algoritmos e no pós-processamento de padrões descobertos para encontrar outros que poderiam ser úteis. Também pode haver mais de um tipo de padrão que pode ser descoberto a partir de determinado banco de dados, e a interação manual pode ser necessária para apanhar tipos úteis de padrões. Por esse motivo, a mineração de dados é na realidade um processo semi-automático na vida real. Porém, em nossa descrição, nos concentramos no aspecto automático da mineração.

Aplicações da mineração de dados

O conhecimento descoberto possui várias aplicações. As aplicações mais utilizadas são aquelas que exigem algum tipo de **previsão**. Por exemplo, quando uma pessoa se candidata a um cartão de crédito, a companhia deseja prever se a pessoa tem um bom crédito. A previsão deve ser baseada em atributos conhecidos da pessoa, como idade, renda, débitos e histórico de negociação de débito. As regras para fazer a previsão são derivadas dos mesmos atributos dos proprietários de cartão passados e atuais, junto com seu comportamento observado, como se eles não pagaram suas dívidas com cartão de crédito. Outros tipos de previsão incluem a previsão de quais clientes podem passar para um concorrente (esses clientes podem receber descontos especiais para instigá-los a não trocar), a previsão de quais pessoas provavelmente responderão ao correio promocional (mala-direta), ou a previsão de quais tipos de uso de cartão de ligação telefônica provavelmente são fraudulentos.

Outra classe de aplicações procura **associações**, por exemplo, livros que costumam ser comprados juntos. Se um cliente comprar um livro, uma livraria on-line poderá sugerir outros livros associados. Se uma pessoa compra uma câmera fotográfica, o sistema pode sugerir acessórios que costumam ser comprados junto com as câmeras. Um bom vendedor conhece bem esses padrões e os explora para fazer vendas adicionais. O desafio é automatizar o processo. Outros tipos de associações podem levar à descoberta de causas. Por exemplo, a descoberta de associações não esperadas entre um remédio recém-introduzido e problemas cardíacos pode fazer com que se descubra que o remédio pode causar problemas cardíacos em algumas pessoas. O remédio, então, pode ser retirado do mercado.

Associações são exemplos de **padrões descritivos**. **Agrupamentos** são outro exemplo desses padrões. Por exemplo, há cerca de um século, um agrupamento de casos de tifo foi encontrado em torno de um poço, o que levou à descoberta de que a água no poço estava contaminada e estava espalhando o tifo. A detecção de agrupamentos de doença continua sendo importante ainda hoje.

Classificação

Como dissemos na seção anterior, a previsão é um dos tipos mais importantes de mineração de dados. Esboçamos o que é classificação, estudamos técnicas para criação de um tipo de classificadores, chamados classificadores de árvore de decisão, e depois estudamos outras técnicas de previsão.

Abstratamente, o problema de **classificação** é este: dado que os itens pertencem a uma dentre várias classes, e dadas instâncias passadas (chamadas **instâncias de treinamento**) dos itens junto com as classes a que pertencem, o problema é prever a classe à qual um novo item pertence. A classe da nova instância não é conhecida, de modo que outros atributos da instância precisam ser usados para prever a classe.

A classificação pode ser feita localizando-se regras que particionam os dados indicados em grupos disjuntos. Por exemplo, suponha que uma empresa de cartão de crédito queira decidir se dará ou não um cartão de crédito a um solicitante. A empresa tem diversas informações sobre a pessoa, como sua idade, grau de instrução, receita anual e débitos atuais, que poderá usar para tomar uma decisão.

Parte dessa informação poderia ser relevante para o merecimento de crédito do solicitante, enquanto parte pode não ser. Para tomar a decisão, a empresa atribui um nível de merecimento de crédito que pode ser excelente, bom, médio ou ruim a cada um de um conjunto de clientes *atuais*, de acordo com o histórico de pagamento do cliente. Depois, a empresa

tenta encontrar regras que classificam seus clientes atuais em excelente, bom, médio ou ruim, com base na informação sobre a pessoa, que não seja o histórico atual de pagamentos (que não está disponível para clientes novos). Vamos considerar apenas dois atributos: grau de instrução (mais alto diploma adquirido) e renda. As regras podem ser da seguinte forma:

$$\forall \text{ pessoa } P,\ P.diploma = mestre \textbf{ and } P.renda > 75000 \Rightarrow P.crédito = excelente$$

$$\forall \text{ pessoa } P,\ P.diploma = bacharel \text{ or } (P.renda \geq 25000 \textbf{ and } P.renda \leq 75000) \Rightarrow P.crédito = bom$$

Regras semelhantes também estariam presentes para outros níveis de merecimento de crédito (médio e ruim).

O processo de criação de um classificador começa com uma amostra de dados, chamada **conjunto de treinamento**. Para cada tupla no conjunto de treinamento, a classe à qual a tupla pertence já é conhecida. Por exemplo, o conjunto de treinamento para uma solicitação de cartão de crédito pode ser dos clientes existentes, com seu merecimento de crédito determinado por seu histórico de pagamentos. Os dados reais, ou população, podem consistir em todas as pessoas, incluindo aqueles que não são clientes existentes. Existem várias maneiras de se criar um classificador, conforme veremos.

Classificadores de árvore de decisão

O classificador de árvore de decisão é uma técnica bastante utilizada para a classificação. Como o nome indica, **classificadores de árvore de decisão** utilizam uma árvore; cada nó de folha possui uma classe associada, e cada nó interno possui um predicado (ou, geralmente, uma função) associado a ele. A Figura 18.8 mostra um exemplo de uma árvore de decisão.

Para classificar uma nova instância, começamos na raiz e atravessamos a árvore até alcançar uma folha; em um nó interno, avaliamos o predicado (ou função) na instância de dados, a fim de descobrir para qual filho prosseguir. O processo continua até que alcancemos um nó de folha. Por exemplo, se o nível de diploma de uma pessoa for mestre, e a receita da pessoa for 40K, começando da raiz, seguimos a linha rotulada com "mestre" e, a partir de lá, a linha rotulada com "25K a 75K", até alcançar uma folha. A classe na folha é "bom", de modo que prevemos que o risco de crédito dessa pessoa é bom.

Criando classificadores de árvore de decisão

A questão, portanto, é como criar um classificador de árvore de decisão, dado um conjunto de instâncias de treinamento. A maneira mais comum de fazer isso é usar um algoritmo **guloso** que funciona recursivamente, começando na raiz e descendo pela árvore. Inicialmente, existe apenas um nó, a raiz, e todas as instâncias de treinamento estão associadas a esse nó.

Em cada nó, se todas ou "quase todas" as instâncias de treinamento associadas ao nó pertencerem à mesma classe, então o nó se torna um nó de folha associado a essa classe. Caso contrário, um **atributo de particionamento** e **condições de particionamento** precisam ser selecionados para criar nós filhos. Os dados associados a cada nó filho é o conjunto de instâncias de treinamento que satis-

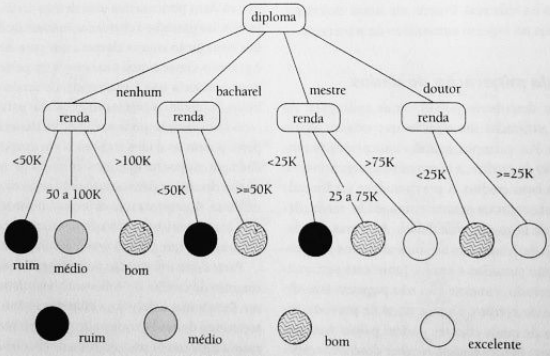


Figura 18.8 Árvore de classificação.

fazem a condição de particionamento para esse nó filho. Em nosso exemplo, o atributo *diploma* é escolhido, e quatro filhos são criados, um para cada valor de diploma. As condições para os quatro nós filhos são *diploma* = nenhum, *diploma* = bacharel, *diploma* = mestre e *diploma* = doutor, respectivamente. Os dados associados a cada filho consistem em instâncias de treinamento satisfazendo a condição associada a cada filho. No nó correspondente a mestre, o atributo *renda* é escolhido, com o intervalo de valores partido em intervalos de 0 a 25K, 25K a 50K, 50K a 75K, e acima de 75K. Os dados associados a cada nó consistem em instâncias de treinamento com o atributo *diploma* sendo mestre e o atributo *renda* estando em cada um desses intervalos, respectivamente. Como uma otimização, visto que a classe para o intervalo de 25K a 50K e o intervalo de 50K a 75K é o mesmo sob o nó *diploma* = mestre, os dois intervalos foram mesclados em um único intervalo de 25K a 75K.

Melhores divisões

Intuitivamente, escolhendo uma seqüência de atributos de particionamento, começamos com o conjunto de todas as instâncias de treinamento, que é "impuro" no sentido de que contém instâncias de muitas classes, e acaba com folhas que são "puras" no sentido de que, em cada folha, todas as instâncias de treinamento pertencem a apenas uma classe. Logo veremos como medir a pureza quantitativamente. Para julgar o benefício de apanhar um atributo e condição em particular para particionamento dos dados em um nó, medimos a pureza dos dados nos filhos resultantes do particionamento por esse atributo. O atributo e a condição que resultam na máxima pureza são escolhidos.

A pureza de um conjunto S de instâncias de treinamento pode ser medida quantitativamente de várias maneiras. Suponha que existam k classes, e das instâncias de S, a fração de instâncias na classe i seja p_i. Uma medida de pureza, a **medida de Gini**, é definida como

$$\text{Gini}(S) = 1 - \sum_{i=1}^{k} p_i^2$$

Quando todas as instâncias estão em uma única classe, o valor de Gini é 0, enquanto alcança seu máximo (de $1 - 1/k$) se cada classe tiver o mesmo número de instâncias. Outra medida de pureza é a **medida de entropia**, que é definida como

$$\text{Entropia}(S) = -\sum_{i=1}^{k} p_i \log_2 p_i$$

O valor de entropia é 0 se todas as instâncias estiverem em uma única classe, e alcança seu máximo quando cada classe tem o mesmo número de instâncias. A medida de entropia é derivada da teoria da informação.

Quando um conjunto S é dividido em vários conjuntos S_i, $i = 1, 2, \ldots, r$, podemos medir a pureza do conjunto resultante como:

$$\text{Pureza}(S_1, S_2, \ldots, S_r) = \sum_{i=1}^{k} \frac{|S_i|}{|S|} \text{pureza}(S_i)$$

Ou seja, a pureza é a média ponderada da pureza dos conjuntos S_i. Essa fórmula pode ser usada com a medida de Gini e a medida de entropia de pureza.

O **ganho de informação** devido a uma divisão em particular de S em S_i, $i = 1, 2, \ldots, r$ é, então

$$\text{Ganho\_informação}(S, \{S_1, S_2, \ldots, S_r\}) = \text{pureza}(S) - \text{pureza}(S_1, S_2, \ldots, S_r)$$

As divisões em conjuntos menores são preferíveis às divisões em muitos conjuntos, pois levam a árvores de decisão mais simples e mais significativas. O número de elementos em cada um dos conjuntos S_i também pode ser levado em consideração; caso contrário, se um conjunto S_i tem 0 ou 1 elemento, isso faria grande diferença no número de conjuntos, embora a divisão seja a mesma para quase todos os elementos. O **conteúdo da informação** de uma divisão em particular pode ser definido em termos de entropia como

$$\text{Conteúdo\_informação}(S, \{S_1, S_2, \ldots, S_r\}) = -\sum_{i=1}^{k} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

Tudo isso leva a uma definição: a **melhor divisão** para um atributo é aquela que dá o máximo de **razão de ganho de informação**, definida como

$$\frac{\text{Ganho\_ informação}(S, \{S_1, S_2, \ldots, S_r\})}{\text{Conteúdo\_ informação}(S, \{S_1, S_2, \ldots, S_r\})}$$

Encontrando as melhores divisões

Como encontramos a melhor divisão para um atributo? Como dividir um atributo depende do tipo do atributo. Os atributos podem ter **valor contínuo**, ou seja, os valores podem ser ordenados de uma maneira significativa à classificação, como idade ou renda, ou podem ser por **categoria**; ou seja, eles não possuem uma ordem significativa, como nomes de departamento ou nomes de país. Não devemos esperar que a ordem de classificação de nomes de departamento ou nomes de país tenha qualquer significado para a classificação.

Normalmente, os atributos que são números (inteiros/reais) são tratados como valores contínuos, enquanto os atributos de string de caracteres são tratados como categorias, mas isso pode ser controlado pelo usuário do siste-

ma. Em nosso exemplo, tratamos o atributo *diploma* como categoria, e o atributo *renda* como valor contínuo.

Primeiro, consideramos como encontrar as melhores divisões para atributos de valor contínuo. Para simplificar, vamos considerar apenas **divisões binárias** de atributos com valor contínuo, ou seja, divisões que resultam em dois filhos. O caso das **divisões multivias** é mais complicado; veja referências sobre o assunto nas notas bibliográficas deste capítulo.

Para encontrar a melhor divisão binária de um atributo com valor contínuo, primeiro classificamos os valores de atributo nas instâncias de treinamento. Depois, calculamos o ganho de informação obtido pela divisão em cada valor. Por exemplo, se as instâncias de treinamento tiverem valores 1, 10, 15 e 25 para um atributo, os pontos de divisão considerados são 1, 10 e 15; em cada caso, os valores menores ou iguais ao ponto de divisão formam uma partição, e o restante dos valores forma a outra partição. A melhor divisão binária para o atributo é a divisão que oferece o máximo de ganho de informação.

Para um atributo de categoria, podemos ter uma divisão multivias, com um filho para cada valor do atributo. Isso funciona bem para atributos de categoria com somente alguns valores distintos, como diploma ou sexo. Porém, se o atributo tiver muitos valores distintos, como nomes de departamento em uma grande empresa, a criação de um filho para cada valor não é uma boa idéia. Nesse caso, tentaríamos combinar diversos valores em cada filho, para criar um número menor de filhos. Veja informações sobre como fazer isso nas notas bibliográficas.

Algoritmo de construção da árvore de decisão

A idéia principal da construção da árvore de decisão é avaliar diferentes atributos e diferentes condições de particionamento, e escolher o atributo e condição de particionamento que resulta no máximo de razão de ganho de informação. O mesmo procedimento funciona recursivamente em cada um dos conjuntos resultantes da divisão, construindo assim recursivamente uma árvore de decisão. Se os dados puderem ser classificados perfeitamente, a recursão pára quando a pureza de um conjunto é 0. Porém, normalmente os dados possuem ruído, ou um conjunto pode ser tão pequeno que seu maior particionamento pode não ser justificado estatisticamente. Nesse caso, a recursão pára quando a pureza de um conjunto é "suficientemente alta" e a classe da folha resultante é definida como a classe da maioria dos elementos do conjunto. Em geral, diferentes ramos da árvore poderiam crescer para diferentes níveis.

A Figura 18.9 mostra o pseudocódigo para um procedimento recursivo de construção de árvore, que apanha um conjunto de instâncias de treinamento S como parâmetro. A recursão pára quando o conjunto é suficientemente puro ou o conjunto S é muito pequeno para que outro particionamento seja estatisticamente significativo. Os parâmetros δ_p e δ_s definem limites para pureza e tamanho; o sistema pode indicar valores-padrão, que podem ser modificados pelos usuários.

Existe uma grande variedade de algoritmos de construção de árvore de decisão, e esboçamos os recursos que distinguem alguns deles. Veja os detalhes nas notas bibliográficas. Com conjuntos de dados muito grandes, o particionamento pode ser dispendioso, pois envolve a cópia repetida. Vários algoritmos, portanto, têm sido desenvolvidos para minimizar a E/S e o custo de computação quando os dados de treinamento são maiores do que a memória disponível.

Vários dos algoritmos também podam subárvores da árvore de decisão gerada para reduzir o sobreajuste: uma su-

```
procedure GrowTree(S)
    Partition(S);

procedure Partition (S)
    if (pureza(S) > δp or |S|< δs ) then
        return;
    for each atributo A
        avalia divisões sobre atributo A;
    Usa melhor divisão achada (por todos os
        atributos) para
            partir S em S1, S2, ..., Sr;
    for i = 1, 2, ..., r
        Partition(Si);
```

Figura 18.9 Construção recursiva de uma árvore de decisão.

árvore é sobreajustada se tiver sido tão sintonizada aos detalhes dos dados de treinamento que criam muitos erros de classificação sobre outros dados. Uma subárvore é podada substituindo-a por um nó de folha. Existem diferentes heurísticas de poda; uma heurística utiliza parte dos dados de treinamento para construir a árvore e outra parte dos dados de treinamento para testá-la. A heurística poda uma subárvore se descobrir que o erro de classificação sobre as instâncias de teste seria reduzido se a subárvore fosse substituída por um nó de folha.

Podemos gerar regras de classificação a partir de uma árvore de decisão, se desejarmos isso. Para cada folha, geramos uma regra como esta: o lado esquerdo é a conjunção de todas as condições de divisão no caminho até a folha, e a classe é a classe da maioria das instâncias de treinamento na folha. Um exemplo dessa regra de classificação é

diploma = *mestre* **and** *renda* > 75000 ⇒ *excelente*

Outros tipos de classificadores

Existem vários tipos de classificadores fora os classificadores de árvore de decisão. Dois tipos que foram muito úteis são os *classificadores de rede neural* e os *classificadores Bayesianos*. Os classificadores de rede neural utilizam os dados de treinamento para treinar redes neurais artificiais. Há uma grande quantidade de literatura sobre redes neurais, e não explicaremos mais sobre elas aqui.

Classificadores Bayesianos encontram a distribuição de valores de atributo para cada classe nos dados de treinamento; quando recebem uma nova instância d, eles usam a informação de distribuição para estimar, para cada classe c_j, a probabilidade de que a instância d pertença à classe c_j, indicada por $p(c_j|d)$, de uma maneira esboçada aqui. A classe com o máximo de probabilidade se torna a classe prevista para a instância d.

Para descobrir a probabilidade $p(c_j|d)$ de a instância d estar na classe c_j, os classificadores bayesianos utilizam o **teorema de Bayes**, que diz

$$p(c_j|d) = \frac{p(d|c_j)p(c_j)}{p(d)}$$

onde $p(d|c_j)$ é a probabilidade de gerar a instância d dada a classe c_j, $p(c_j)$ é a probabilidade de ocorrência da classe c_j, e $p(d)$ é a probabilidade de a instância d ocorrer. Destas, $p(d)$ pode ser ignorada, pois é a mesma para todas as classes. $p(c_j)$ é simplesmente a fração de instâncias de treinamento que pertencem à classe c_j.

É difícil encontrar $p(d|c_j)$ com exatidão, pois exige uma distribuição completa de instâncias de c_j. Para simplificar a tarefa, **classificadores Bayesianos simples** consideram que os atributos possuem distribuições independentes, e portanto estimam

$$p(d|c_j) = p(d_1|c_j) * p(d_2|c_j) * \ldots * p(d_n|c_j)$$

Ou seja, a probabilidade de a instância d ocorrer é o produto da probabilidade de ocorrência de cada um dos valores de atributo d_i de d, dado que a classe é c_j.

As probabilidades $p(d_i|c_j)$ derivam da distribuição de valores para cada atributo i, para cada classe c_j. Essa distribuição é calculada a partir das instâncias de treinamento que pertencem a cada classe c_j; a distribuição normalmente é aproximada por um histograma. Por exemplo, podemos dividir o intervalo de valores do atributo i em intervalos iguais, e armazenar a fração de instâncias da classe c_j que cairá em cada intervalo. Dado um valor d_i para o atributo i, o valor de $p(d_i|c_j)$ é simplesmente a fração de instâncias pertencentes à classe c_j que caem no intervalo ao qual d_i pertence.

Um benefício significativo dos classificadores bayesianos é que eles podem classificar instâncias com valores com atributo desconhecido e nulo – atributos desconhecidos ou nulos são simplesmente omitidos do cálculo da probabilidade. Ao contrário, os classificadores de árvore de decisão não podem tratar de forma significativa situações em que uma instância a ser classificada possui um valor nulo para um atributo de particionamento usado para atravessar melhor a árvore de decisão.

Regressão

A regressão lida com a previsão de um valor, em vez de uma classe. Dados valores para um conjunto de variáveis, $X_1, X_2, \ldots, X_n$, queremos prever o valor de uma variável Y. Por exemplo, poderíamos tratar o grau de instrução como um número e a renda como outro número e, com base nessas duas variáveis, queremos prever a probabilidade de inadimplência, que poderia ser uma chance percentual de inadimplência, ou a quantidade envolvida na inadimplência.

Uma maneira é inferir coeficientes $a_0, a_1, a_2, \ldots, a_n$, de modo que

$$Y = a_0 + a_1 * X_1 + a_2 * X_2 + \ldots + a_n * X_n$$

Localizar um polinômio linear é chamado de **regressão linear**. Em geral, queremos encontrar uma curva (definida por um polinômio ou outra fórmula) que caiba os dados; o processo também é chamado **ajuste de curva**.

O ajuste pode ser apenas aproximado, devido ao ruído nos dados ou porque o relacionamento não é exatamente um polinômio, de modo que a regressão visa encontrar coeficientes que dão o melhor ajuste possível. Existem técnicas padronizadas em estatística para encontrar os coeficientes de regressão. Não discutimos essas técnicas aqui, mas as notas bibliográficas contêm referências.

Regras de associação

As lojas de revenda normalmente estão interessadas em associações entre diferentes itens que as pessoas compram. Alguns exemplos dessas associações são:

- Alguém que compra pão provavelmente também comprará leite.
- Uma pessoa que comprou o livro *Conceitos de Sistema de Banco de Dados* provavelmente também comprará o livro *Conceitos de Sistema Operacional*.

A informação de associação pode ser usada de várias maneiras. Quando um cliente compra determinado livro, uma loja on-line pode sugerir livros associados. Um mercado pode decidir colocar o pão perto do leite, pois eles normalmente são comprados juntos, para que os compradores terminem sua tarefa mais rapidamente. Ou então a loja pode colocá-los em cantos opostos de uma prateleira, colocando outros itens associados no intervalo, para que as pessoas também comprem esses itens, enquanto vão de um canto da prateleira ao outro. Uma loja que oferece descontos sobre um item associado pode não oferecer um desconto sobre o outro, pois o cliente provavelmente comprará o outro, de qualquer forma.

Um exemplo de uma regra de associação é

$$pão \Rightarrow leite$$

No contexto de compras de supermercado, a regra diz que os clientes que compram pão também costumam comprar leite com uma alta probabilidade. Uma regra de associação precisa ter uma **população** associada. A população consiste em um conjunto de **instâncias**. No exemplo de supermercado, a população pode consistir em todas as compras do mercado; cada compra é uma instância. No caso de uma livraria, a população pode consistir em todas as pessoas que fizeram compras, independente de quando elas compraram. Cada cliente é uma instância. Aqui, o analista decidiu que *quando* uma compra é feita não é algo significativo, enquanto para o exemplo de mercado, o analista pode ter decidido se concentrar em compras isoladas, ignorando múltiplas visitas pelo mesmo cliente.

As regras possuem um *suporte* associado, bem como uma *confiança* associada. Estes são definidos no contexto da população:

- **Suporte** é uma medida de qual fração da população satisfaz o antecedente e o conseqüente da regra.

  Por exemplo, suponha que apenas 0,001% de todas as compras incluam leite e parafusos. O suporte para a regra

  $$pão \Rightarrow parafusos$$

  é baixo. A regra pode nem sequer ser estatisticamente significativa – talvez houvesse apenas uma única compra que incluísse tanto leite quanto parafusos. As empresas normalmente não estão interessadas em regras que possuem baixo suporte, pois envolvem poucos clientes e não merecem atenção.

  Por outro lado, se 50% de todas as compras envolverem leite e pão, então o suporte para as regras envolvendo pão e leite (e nenhum outro item) é relativamente alto, e essas regras podem merecer atenção. Exatamente que grau mínimo de suporte é considerado desejável depende da aplicação.
- **Confiança** é uma medida da freqüência com que o conseqüente é verdadeiro quando o antecedente é verdadeiro. Por exemplo, a regra

  $$pão \Rightarrow leite$$

  tem uma confiança de 80% se 80% das compras que incluem pão também incluírem leite. Uma regra com uma baixa confiança não é significativa. Em aplicações de negócios, as regras normalmente possuem confianças significativamente menores que 100%, enquanto em outros domínios, como na física, as regras podem ter confianças altas.

  Observe que a confiança de $pão \Rightarrow leite$ pode ser muito diferente da confiança de $leite \Rightarrow pão$, embora ambas tenham o mesmo suporte.

Para descobrir as regras de associação na forma

$$i_1, i_2, \ldots, i_n \Rightarrow i_0$$

primeiro encontramos conjuntos de itens com suporte suficiente, chamados **itemsets grandes**. Em nosso exemplo, encontramos conjuntos de itens que estão incluídos em um número suficientemente grande de instâncias. Logo veremos como calcular itemsets grandes.

Para cada itemset grande, enviamos todas as regras com confiança suficiente que envolvam todos e somente os elementos do conjunto. Para cada itemset S grande, enviamos uma regra $S - s \Rightarrow s$ para cada subconjunto $s \subset S$, desde que $S - s \Rightarrow s$ tenha confiança suficiente; a confiança da regra é dada pelo suporte de s dividido pelo suporte de S.

Agora, consideramos como gerar todos os itemsets grandes. Se o número de conjuntos de itens possíveis for pequeno, uma única passada pelos dados é suficiente para detectar o nível de suporte para todos os conjuntos. Um contador, inicializado com 0, é mantido para cada conjunto de itens. Quando um registro de compra é apanhado, o contador é incrementado para cada conjunto de itens de modo que todos os itens no conjunto estejam contidos na compra. Por exemplo, se uma compra incluísse os itens *a*, *b* e *c*,

os contadores seriam incrementados para {a}, {b}, {c}, {a, b}, {b,c}, {a, c} e {a, b, c}. Aqueles conjuntos com um contador suficientemente alto no final da passada correspondem aos itens que possuem um alto grau de associação.

O número de conjuntos cresce exponencialmente, tornando o procedimento recém-descrito inviável se o número de itens for grande. Felizmente, quase todos os conjuntos em geral teriam suporte muito baixo; as otimizações foram desenvolvidas para eliminar a maioria desses conjuntos de consideração. Essas técnicas utilizam várias passadas no banco de dados, considerando apenas alguns conjuntos em cada passada.

Na técnica **a priori** para gerar itemsets grandes, somente conjuntos com itens isolados são considerados na primeira passada. Na segunda passada, conjuntos com dois itens são considerados, e assim por diante.

Ao final de uma passada, todos os conjuntos com suporte suficiente são enviados como grandes itemsets. Os conjuntos que possuem muito pouco suporte no final de uma passada são eliminados. Quando um conjunto é eliminado, nenhum de seus superconjuntos precisa ser considerado. Em outras palavras, na passada i, precisamos contar apenas suportes para conjuntos de tamanho i tal que todos os subconjuntos do conjunto tenham suporte suficientemente alto; basta testar todos os subconjuntos de tamanho $i - 1$ para garantir essa propriedade. Ao final de alguma passada i, descobriríamos que nenhum conjunto de tamanho i possui suporte suficiente, de modo que não precisamos considerar qualquer conjunto de tamanho $i + 1$. Nesse ponto, a computação termina.

Outros tipos de associações

O uso de regras de associação comuns possui várias desvantagens. Uma das maiores é que muitas associações não são muito interessantes, pois podem ser previstas. Por exemplo, se muitas pessoas comprarem cereal e muitas pessoas comprarem pão, podemos prever que um número muito grande de pessoas compraria ambos, mesmo que não haja conexão entre as duas compras. De fato, mesmo que a compra de cereal tenha uma influência negativa moderada na compra de pão (ou seja, os clientes que compram cereal tendem a comprar pão com menos freqüência do que o cliente normal), a associação entre cereal e pão ainda pode ter um alto suporte.

O que seria mais interessante é um **desvio** da co-ocorrência esperada dos dois. Em termos estatísticos, procuramos **correlações** entre itens; as correlações podem ser positivas, no sentido de que a co-ocorrência é mais alta do que se esperaria, ou negativas, no sentido de que os itens co-ocorrem com menos freqüência do que o previsto. Assim, se a compra de pão não estiver correlacionada com cereal, ela não seria informada, mesmo que houvesse uma associação forte entre os dois. Existem medidas-padrão de correlação, muito usadas na área de estatística. Para obter mais informações sobre correlações, consulte um livro-texto padrão sobre estatística.

Outra classe importante de aplicações de mineração de dados é a de associações em seqüência (ou correlações em seqüência). Os dados de série de tempo, como preços de ações em uma seqüência de dias, formam um exemplo de dados de seqüência. Os analistas do mercado de ações querem encontrar associações entre as seqüências de preços do mercado de ações. Um exemplo desse tipo de associação é a seguinte regra: "Sempre que as taxas de juros sobem, os preços da ação descem dentro de 2 dias". A descoberta de tal associação entre sequências pode nos ajudar a tomar decisões inteligentes sobre investimento. Consulte as notas bibliográficas para obter referências a pesquisas sobre o assunto.

Os desvios de padrões temporais normalmente são interessantes. Por exemplo, se uma empresa tiver crescido em uma taxa constante a cada ano, um desvio da taxa de crescimento normal é uma surpresa. Se as vendas de roupas de inverno cairem no verão, isso não é surpresa, pois podemos prever pelos anos anteriores; um desvio que não poderíamos ter previsto da experiência do passado seria considerado interessante. As técnicas de mineração podem encontrar desvios a partir do que alguém teria esperado com base nos padrões temporais/seqüenciais do passado. Consulte as notas bibliográficas para obter referências à pesquisa sobre o assunto.

Agrupamento

Intuitivamente, o agrupamento refere-se ao problema de localizar clusters de pontos nos dados indicados. O problema de **agrupamento** pode ser formalizado a partir de medidas de distância de várias maneiras. Uma maneira é indicá-lo como o problema de agrupar pontos em k conjuntos (para um k qualquer indicado) de modo que a distância média dos pontos a partir do *centróide* do seu cluster atribuído seja reduzida.[5] Outra maneira é agrupar pontos de modo que a distância média entre cada par de pontos em cada cluster seja reduzida. Também existem outras definições; veja os detalhes nas notas bibliográficas. No entanto, a intuição por trás de todas essas definições é agrupar pontos semelhantes em um único conjunto.

5. O centróide de um conjunto de pontos é definido como um ponto cuja coordenada em cada dimensão é a média das coordenadas de todos os pontos desse conjunto nessa dimensão. Por exemplo, em duas dimensões, o centróide de um conjunto de pontos $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ é dado por $\left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n}\right)$.

Outro tipo de agrupamento aparece nos sistemas de classificação na biologia. (Esses sistemas de classificação não tentam *prever* classes; em vez disso, eles tentam manter itens relacionados em clusters.) Por exemplo, leopardos e humanos são agrupados sob a classe mamíferos, enquanto crocodilos e serpentes são agrupados sob répteis. Tanto mamíferos quanto répteis vêm abaixo da classe comum cordato. O agrupamento de mamíferos possui ainda outros subclusters, como carnívoros e primatas. Assim, temos o **agrupamento hierárquico**. Dadas as características de diferentes espécies, os biologos criaram um esquema de agrupamento hierárquico complexo, agrupando espécies relacionadas em diferentes níveis de hierarquia.

O agrupamento hierárquico também é útil em outros domínios – para agrupar documentos, por exemplo. Os sistemas de diretório da Internet (como o do Yahoo) agrupam documentos relacionados em um padrão hierárquico (ver seção "Diretórios" do Capítulo 19). Os algoritmos de agrupamento hierárquico podem ser classificados como algoritmo de **agrupamento aglomerado**, que começam montando pequenos clusters e depois criam níveis mais altos, ou algoritmos de **agrupamento divisível**, que primeiro criam níveis mais altos do agrupamento hierárquico, depois refinam cada cluster resultante em clusters de nível inferior.

A comunidade de estatística tem estudado muito sobre os agrupamentos. A pesquisa em banco de dados tem oferecido algoritmos de agrupamento escaláveis, que podem agrupar conjuntos de dados muito grandes (que podem não caber na memória). O algoritmo de agrupamento de Birch é um algoritmo desse tipo. Intuitivamente, pontos de dados são inseridos em uma estrutura de árvore multidimensional (baseados em árvores-R, descritos na seção "Árvores R" do Capítulo 24), e guiados para os nós de folha apropriados com base na proximidade com pontos representativos nos nós internos da árvore. Pontos vizinhos, assim, são agrupados em nós de folha e resumidos se houver mais pontos do que pode caber na memória. O resultado dessa primeira fase de agrupamento é criar um conjunto de dados parcialmente agrupado, que caiba na memória. As técnicas de agrupamento padrão podem, então, ser executadas nos dados da memória para chegar ao agrupamento final. Veja nas notas bibliográficas referências ao algoritmo de Birch e outras técnicas para agrupamento, incluindo algoritmos para agrupamento hierárquico.

Uma aplicação interessante do agrupamento é prever quais novos filmes (ou livros, ou músicas) uma pessoa provavelmente estará interessada, com base:

1. Na preferência passada dessa pessoa pelos filmes
2. Em outras pessoas com preferências passadas semelhantes
3. Nas preferências de tais pessoas por novos filmes

Uma solução para esse problema é a seguinte. Para encontrar pessoas com preferências passadas semelhantes, criamos clusters de pessoas com base em suas preferências por filmes. A precisão do agrupamento pode ser melhorada por filmes agrupados anteriormente por sua semelhança, de modo que, se as pessoas não tiverem visto os mesmos filmes, mas se tiverem visto filmes semelhantes, elas seriam agrupadas juntas. Podemos repetir o agrupamento, alternadamente agrupando pessoas, depois filmes, depois pessoas, e assim por diante, até alcançarmos um equilíbrio. Dado um novo usuário, encontramos um cluster de usuários mais semelhantes a esse usuário, com base em suas preferências por filmes já vistos. Depois, prevemos filmes nos clusters de filmes que são populares com o cluster desse usuário como provavelmente sendo interessantes para o novo usuário. De fato, esse problema é uma instância de *filtragem colaborativa*, em que os usuários colaboram na tarefa de filtrar informações a fim de encontrar informações de interesse.

Outros tipos de exploração

A **exploração de texto** se aplica a técnicas de mineração de dados aos documentos textuais. Por exemplo, existem ferramentas que formam clusters sobre páginas que um usuário visitou; isso ajuda os usuários quando eles navegam pelo histórico de sua navegação para encontrar páginas que visitaram anteriormente. A distância entre as páginas pode ser baseada, por exemplo, em palavras comuns nas páginas (ver seção "Recuperação baseada em semelhança" do Capítulo 19). Outra aplicação é classificar as páginas em um diretório da Web automaticamente, de acordo com sua semelhança com outras páginas (ver seção "Diretórios" do Capítulo 19).

Os sistemas de **visualização de dados** ajudam os usuários a examinar grandes volumes de dados e detectar padrões visualmente. Apresentações visuais dos dados – como mapas, diagramas e outras representações gráficas – permitem que os dados sejam apresentados de forma compacta aos usuários. Uma única tela gráfica pode codificar tanta informação quanto um número muito maior de telas de texto. Por exemplo, se o usuário deseja descobrir se os problemas da produção nas fábricas estão relacionados aos locais das fábricas, os locais problemáticos podem ser codificados em uma cor especial – digamos, vermelha – em um mapa. O usuário pode, então, descobrir rapidamente os locais onde os problemas estão ocorrendo. O usuário pode, então, formar hipóteses sobre o motivo para os problemas estarem ocorrendo nesses locais, e pode verificar as hipóteses quantitativamente contra o banco de dados.

Como outro exemplo, as informações sobre os valores podem ser codificadas como uma cor, e podem ser exibidas com até mesmo um pixel de área da tela. Para detectar asso-

ciações entre pares de itens, podemos usar uma matriz de pixels bidimensional, com cada linha e cada coluna representando um item. A porcentagem de transações que compram os dois itens pode ser codificada pela intensidade de cor do pixel. Os itens com alta associação aparecerão como pixels claros na tela – fáceis de detectar contra o fundo mais escuro.

Os sistemas de visualização de dados não detectam padrões automaticamente, mas oferecem suporte do sistema para os usuários detectarem padrões. Como os humanos são muito bons na detecção de padrões visuais, a visualização de dados é um componente importante da mineração de dados.

Resumo

- Os sistemas de apoio à decisão analisam dados on-line coletados por sistemas de processamento de transação, para ajudar as pessoas a tomarem decisões de negócios. Como a maioria das organizações é bastante informatizada hoje, uma grande quantidade de informação está disponível para apoio à decisão. Os sistemas de apoio à decisão têm diversos formatos, incluindo sistemas OLAP e sistemas de mineração de dados.
- Ferramentas de processamento analítico on-line (OLAP) ajudam os analistas a verem os dados resumidos de diferentes maneiras, de modo que possam discernir a respeito do funcionamento de uma organização.
  - Ferramentas OLAP funcionam sobre dados multidimensionais, caracterizados por atributos de dimensão e atributos de medida.
  - O cubo de dados consiste em dados multidimensionais resumidos de diferentes maneiras. O cálculo prévio do cubo de dados ajuda a agilizar consultas sobre resumos de dados.
  - Visões de tabulação cruzada permitem que os usuários vejam duas dimensões dos dados multidimensionais de uma só vez, junto com resumos dos dados.
  - Descer, subir, slicing e dicing estão entre as operações que os usuários realizam com ferramentas OLAP.
- O componente OLAP do padrão SQL:1999 oferece uma série de novas funcionalidades para análise de dados, incluindo novas funções agregadas; operações cube e rollup; funções de classificação; funções de janela, que admitem resumo em janelas móveis; e particionamento, com janelas e classificação aplicados dentro de cada partição.
- Depósitos de dados ajudam a reunir e arquivar dados operacionais importantes. Depósitos são usados para apoio à decisão e análise sobre dados históricos, por exemplo, para prever tendências. A limpeza de dados a partir das fontes de dados de entrada normalmente é uma tarefa importante no depósito de dados. Os esquemas de depósito costumam ser multidimensionais, envolvendo uma ou algumas tabelas de fatos muito grandes, e várias tabelas de dimensão muito menores.
- Mineração de dados é o processo de analisar de maneira semi-automática grandes bancos de dados para encontrar padrões úteis. Existem diversas aplicações da mineração de dados, como a previsão de valores com base em exemplos passados, descoberta de associações entre compras e agrupamento automático de pessoas e filmes.
- A classificação trata da previsão da classe de instâncias de teste, usando atributos de instâncias de teste, com base nos atributos das instâncias de treinamento, e a classe real de instâncias de treinamento. A classificação pode ser usada, por exemplo, para prever níveis de validade de crédito de novos candidatos ou para prever o desempenho de candidatos a uma universidade.

  Existem vários tipos de classificadores, como
  - Os classificadores de árvore de decisão, que realizam a classificação construindo uma árvore baseada nas instâncias de treinamento com folhas tendo rótulos de classe. A árvore é atravessada para cada instância de teste a fim de encontrar uma folha, e a classe da folha é a classe prevista. Várias técnicas estão disponíveis para construir árvores de decisão, a maioria delas baseada em heurísticas vorazes.
  - Os classificadores bayesianos são mais simples de construir do que os classificadores de árvore de decisão, e funcionam melhor no caso de valores de atributo faltando/nulos.
- As regras de associação identificam itens que co-ocorrem com freqüência, por exemplo, itens que costumam ser comprados pelo mesmo cliente. As correlações procuram desvios dos níveis de associação esperados.
- Outros tipos de mineração de dados incluem agrupamento, exploração de texto e visualização de dados.

Termos de revisão

- Sistemas de apoio à decisão
- Análise estatística
- Dados multidimensionais
  - Atributos de medida
  - Atributos de dimensão
- Tabela cruzada
- Cubo de dados
- Processamento analítico on-line (OLAP)
  - Pivotar
  - Slicing e dicing
  - Subir e descer
- OLAP multidimensional (MOLAP)
- OLAP relacional (ROLAP)

- OLAP híbrido (HOLAP)
- Agregação estendida
  - Variância
  - Desvio-padrão
  - Correlação
  - Regressão
- Funções de classificação
  - Rank
  - Dense rank
  - Partition by
- Janelas
- Depósito de dados
  - Colheita de dados
  - Arquitetura controlada por origem
  - Arquitetura controlada por destino
  - Limpeza de dados
  - Merge-purge
  - Householding
  - Extract, Transform, Load (ETL)
- Esquemas de depósito
  - Tabela de fatos
  - Tabelas de dimensão
  - Esquema de estrela
  - Mineração de dados
- Previsão
- Associações
- Classificação
  - Dados de treinamento
  - Dados de teste
- Classificadores de árvore de decisão
  - Atributo de particionamento
  - Condição de particionamento
  - Pureza
  - Medida de Gini
  - Medida de entropia
  - Ganho de informação
  - Conteúdo de informação
  - Razão de ganho de informação
  - Atributo de valor contínuo
  - Atributo de categoria
  - Divisão binária
  - Divisão multivias
  - Sobreajuste
- Classificadores de Bayes
  - Teorema de Bayes
  - Classificadores bayesianos simples
- Regressão
  - Regressão linear
  - Ajuste de curva
- Regras de associação
  - População
  - Suporte
  - Confiança
  - Itemsets grandes
- Outros tipos de associações
- Agrupamento
  - Agrupamento hierárquico
  - Agrupamento aglomerado
  - Agrupamento divisível
- Exploração de texto
- Visualização de dados

Exercícios práticos

18.1 Mostre como expressar **group by cube**(a, b, c, d) usando rollup; sua resposta deverá ter apenas uma cláusula group by.

18.2 Dada uma relação S(*aluno, matéria, notas*), escreva uma consulta para encontrar os n primeiros alunos por total de notas, usando a classificação.

18.3 Escreva uma consulta para encontrar saldos cumulativos, equivalente à que aparece na seção "Agregação estendida", mas sem usar as construções estendidas de janela SQL.

18.4 Considere a relação *vendas* da seção "Análise de dados e OLAP". Escreva uma consulta SQL para calcular a operação de cubo sobre a relação, dando a relação na Figura 18.2. Não use a construção **cube**.

18.5 Descreva benefícios e desvantagens de uma arquitetura controlada por origem para colher dados em um depósito de dados, em comparação com uma arquitetura controlada por destino.

18.6 Suponha que existam duas regras de classificação, uma que diz que as pessoas com salários entre \$10.000 e \$20.000 tenham uma avaliação de crédito *bom*, e outra que diz que as pessoas com salários entre \$20.000 e \$30.000 tenham uma avaliação de crédito *bom*. Sob quais condições as regras podem ser substituídas, sem qualquer perda de informação, por uma única regra que diz que as pessoas com salários entre \$10.000 e \$30.000 tenham uma avaliação de crédito *bom*.

18.7 Considere o esquema representado na Figura 18.7. Dê uma consulta SQL:1999 para resumir os números de vendas e o preço por loja e data, junto com as hierarquias sobre loja e data.

Exercícios

18.8 Para cada uma das funções de agregação da SQL **sum**, **count**, **min** e **max**, mostre como calcular o valor de agregação em um multiconjunto $S_1 \cup S_2$, dados os valores de agregação sobre multiconjuntos S_1 e S_2.

Com base nisso, crie expressões para calcular valores de agregação com agrupamento sobre um subconjunto S dos atributos de uma relação $r(A, B, C, D, E)$, dados valores de agregação para agrupamento sobre os atributos $T \supseteq S$, para as seguintes funções de agregação:

a. **sum, count, min** e **max**
b. **avg**
c. Desvio-padrão

18.9 Dê um exemplo de um par de agrupamentos que não podem ser expressos usando uma única cláusula **group by** com **cube** e **rollup**.

18.10 Dada a relação $r(a, b, c)$, mostre como usar os recursos da SQL estendida para gerar um histograma de c *versus* a, dividindo a por 20 partições de mesmo tamanho (ou seja, onde cada partição contém 5% das tuplas em r, classificadas por a).

18.11 Considere o atributo *saldo* da relação *conta*. Escreva uma consulta SQL para calcular um histograma de valores de *saldo*, dividindo o intervalo de 0 até o saldo de conta máximo presente, em três intervalos iguais.

18.12 Construa um classificador de árvore de decisão com divisões binárias em cada nó, usando tuplas na relação $r(A, B, C)$ mostradas a seguir como dados de treinamento; o atributo C indica a classe. Mostre a árvore final, e com cada nó, mostre a melhor divisão para cada atributo junto com seu valor de ganho de informação.

$(1, 2, a), (2, 1, a), (2, 5, b), (3, 3, b),$
$(3, 6, b), (4, 5, b), (5, 5, c), (6, 3, b), (6, 7, c)$

18.13 Suponha que metade de todas as transações em uma loja de roupas seja de compra de jeans, e um terço de todas as transações na loja seja de compra de camisas de malha. Suponha também que metade das transações que de compra de jeans também compre camisas de malha. Escreva todas as regras de associação (não triviais) que você possa deduzir a partir dessa informação, dando o suporte e a confiança de cada regra.

18.14 Considere o problema de encontrar grandes itemsets.

a. Descreva como encontrar o suporte para determinada coleção de itemsets usando uma única varredura dos dados. Considere que os itemsets e as informações associadas, como contas, caberão na memória.
b. Suponha que um itemset tenha suporte menor do que j. Mostre que nenhum superconjunto desse itemset pode ter suporte maior ou igual a j.

18.15 Crie um pequeno exemplo de um conjunto de transações mostrando que, embora muitas transações contenham dois itens, esse é o itemset contendo os dois itens, ou seja, o itemset contendo os dois itens tem um suporte alto, a compra de um dos itens pode ter uma correlação negativa com a compra da outra.

18.16 A organização de partes, capítulos, seções e subseções em um livro está relacionada ao agrupamento. Explique por que e a que forma de agrupamento.

18.17 Sugira como as técnicas de exploração previsíveis podem ser usadas por um time esportivo, usando seu esporte favorito como um exemplo.

Notas bibliográficas

Gray *et al.* [1995] e Gray *et al.* [1997] descrevem o operador data-cube. Algoritmos eficientes para calcular cubos de dados são descritos por Agarwal *et al.* [1996], Harinarayan *et al.* [1996] e Ross e Srivastava [1997]. Descrições de suporte à agregação estendida na SQL:1999 podem ser encontradas no manual de produto dos sistemas de banco de dados, como Oracle e IBM DB2. As definições das funções estatísticas podem ser encontradas nos livros-texto padrão de estatística, como Bulmer [1979] e Ross [1999].

Poe [1995] e Mattison [1996] oferecem um livro-texto sobre depósito de dados (data warehousing). Zhuge *et al.* [1995] descrevem a manutenção de visão em um ambiente de depósito de dados. Chaudhuri *et al.* [2003] descrevem técnicas para combinação difusa para limpeza de dados, enquanto Sarawagi *et al.* [2002] descrevem um sistema para deduplicação usando técnicas de aprendizado ativo.

Witten e Frank [1999] e Han e Kamber [2000] oferecem um livro-texto da mineração de dados. Mitchell [1997] é um livro-texto clássico sobre aprendizado de máquina que aborda técnicas de classificação com detalhes. Fayyad *et al.* [1995] apresentam uma coleção extensa de artigos sobre descoberta de conhecimento e mineração de dados. Kohavi e Provost [2001] apresentam uma coleção de artigos sobre aplicações de mineração de dados no comércio eletrônico.

Agrawal *et al.* [1993b] oferecem uma visão geral da mineração de dados nos bancos de dados. Os algoritmos para calcular classificadores com grandes conjuntos de treinamento são descritos por Agrawal *et al.* [1992] e Shafer *et al.* [1996]; o algoritmo de construção da árvore de decisão descrito neste capítulo é baseado no algoritmo SPRINT de Shafer *et al.* [1996]. Agrawal *et al.* [1993a] introduziram a noção de regras de associação, enquanto Agrawal e Srikant [1994] apresentam um algoritmo eficiente para a exploração da regra de associação. Os algoritmos para mineração de diferentes formas de regras de associação são descritos por Srikant e Agrawal [1996a] e Srikant e Agrawal [1996b].

Chakrabarti *et al.* [1998] descrevem técnicas para mineração de padrões temporais inesperados.

Técnicas para integrar cubos de dados com a mineração de dados são descritas por Sarawagi [2000].

O agrupamento há muito tempo tem sido estudado na área de estatísticas, e Jain e Dubes [1988] oferecem um livro-texto sobre agrupamento. Ng e Han [1994] descrevem técnicas de agrupamento espacial. As técnicas de agrupamento para grandes conjuntos de dados são descritas por Zhang *et al.* [1996]. Breese et al. [1998] oferecem uma análise empírica de diferentes algoritmos para a filtragem colaborativa. As técnicas para filtragem colaborativa de novos artigos são descritas por Konstan *et al.* [1997].

Chakrabarti [2002] oferece um livro-texto sobre recuperação de informações, incluindo uma explicação extensa sobre tarefas de mineração de dados relacionadas a dados textuais e de hipertexto, como classificação e agrupamento. Chakrabarti [2000] oferece um levantamento das técnicas de exploração de hipertexto, como a classificação e o agrupamento de hipertexto.

Ferramentas

Diversas ferramentas estão disponíveis para cada uma das aplicações que estudamos neste capítulo. A maioria dos fornecedores de banco de dados oferece ferramentas OLAP como parte de seu sistema de banco de dados, ou como aplicações adicionais. Estas incluem ferramentas OLAP da Microsoft Corp., Oracle Express e Informix Metacube. A ferramenta OLAP Arbor Essbase é de um fornecedor de software independente. O site `www.databeacon.com` oferece uma demonstração das ferramentas OLAP da Databeacon para uso na Web e em origens de dados de arquivo de texto. Muitas empresas também oferecem ferramentas de análise para aplicações específicas, como gestão de relacionamento com o cliente.

Os principais fornecedores de banco de dados também oferecem produtos de depósito de dados junto com seus sistemas de banco de dados. Estes oferecem funcionalidade de suporte para modelagem, limpeza, carga e consulta de dados. O site `www.dwinfocenter.org` oferece informações sobre produtos de depósito de dados.

Há também uma grande variedade de ferramentas de uso geral para mineração de dados, incluindo ferramentas de exploração do SAS Institute, IBM Intelligent Miner e SGI Mineset. É preciso haver muita habilidade para aplicar ferramentas de exploração de uso geral para aplicações específicas. Como resultado, uma grande quantidade de ferramentas de exploração tem sido desenvolvida para tratar de aplicações especializadas. O site `www.kdnuggets.com` oferece um diretório extenso de software de exploração, soluções, publicações e assim por diante.

CAPÍTULO 19

Recuperação de informações

Os dados textuais são desestruturados, ao contrário dos dados rigidamente estruturados nos bancos de dados relacionais. O termo **recuperação de informações** geralmente se refere à consulta de dados textuais não estruturados. Os sistemas de recuperação de informações têm muito em comum com os sistemas de banco de dados, em particular, o armazenamento e a recuperação de dados no armazenamento secundário. Porém, a ênfase no campo de sistemas de informação é diferente daquele de sistemas de banco de dados, concentrando-se em questões como consulta baseada em palavras-chave; a relevância de documentos à consulta; e a análise, classificação e indexação de documentos.

Visão geral

O campo de **recuperação de informações** se desenvolveu em paralelo com o campo de bancos de dados. No modelo tradicional usado no campo de recuperação de informações, a informação é organizada em documentos, e se considera que existe uma grande quantidade de documentos. Os dados contidos nos documentos não são estruturados, e não dispõem de qualquer esquema associado. O processo de recuperação de informações consiste em localizar documentos relevantes, com base na entrada do usuário, como palavras-chave ou documentos de exemplo.

A Web oferece um modo conveniente de chegar até e interagir com fontes de informação pela Internet. Porém, um problema persistente encarado pela Web é a explosão de informações armazenadas, com pouca orientação para ajudar o usuário a localizar o que é interessante. A recuperação de informações tem desempenhado um papel crítico para tornar a Web uma ferramenta produtiva e útil, especialmente para os pesquisadores.

Exemplos tradicionais de sistemas de recuperação de informações são catálogos de biblioteca on-line e sistemas de gerenciamento de documentos on-line, como aqueles que armazenam artigos de jornal. Os dados nesses sistemas são organizados como uma coleção de *documentos*; um artigo de jornal e uma entrada de catálogo (em um catálogo de biblioteca) são exemplos de documentos. No contexto da Web, normalmente cada página HTML é considerada como um documento.

Um usuário de tal sistema pode querer apanhar um documento em particular ou uma classe específica de documentos. Os documentos intencionados normalmente são descritos por um conjunto de **palavras-chave** – por exemplo, as palavras-chave "sistema de banco de dados" podem ser usadas para localizar livros sobre sistemas de banco de dados, e as palavras-chave "ações" e "escândalo" podem ser usadas para localizar artigos sobre escândalos no mercado de ações. Associado a eles, os documentos possuem um conjunto de palavras-chave, e os documentos cujas palavras-chave contêm termos fornecidos pelo usuário são recuperados.

A recuperação de informações baseada em palavra-chave pode ser usada não apenas para recuperação de dados textuais, mas também para a recuperação de outros tipos de dados, como dados de vídeo e áudio, que possuem palavras-chave descritivas associadas. Por exemplo, um filme de vídeo pode ter, associado a ele, palavras-chave como seu título, diretor, atores e tipo.

Existem várias diferenças entre esse modelo e os modelos usados nos sistemas tradicionais de banco de dados.

- Os sistemas de banco de dados tratam de várias operações que não são consideradas nos sistemas de recupera-

ção de informações. Por exemplo, os sistemas de banco de dados lidam com atualizações e com os requisitos transacionais associados do controle de concorrência e durabilidade. Essas questões são vistas como menos importantes nos sistemas de informação. De modo semelhante, os sistemas de banco de dados lidam com informações estruturadas, organizadas com modelos de dados relativamente complexos (como o modelo relacional ou modelos de dados orientados a objeto), enquanto os sistemas de recuperação de informações tradicionalmente têm usado um modelo muito mais simples, em que a informação no banco de dados é organizada simplesmente como uma coleção de documentos não estruturados.

- Os sistemas de recuperação de informações lidam com várias questões que não foram consideradas adequadamente nos sistemas de banco de dados. Por exemplo, o campo de recuperação de informações lida com os problemas de gerenciamento de documentos não estruturados, como a busca aproximada de palavras-chave, e da classificação de documentos em graus estimados de relevância dos documentos à consulta.

Os sistemas de recuperação de informações normalmente permitem que expressões de consulta formadas usando palavras-chave e os conectivos lógicos *and*, *or* e *not*. Por exemplo, um usuário poderia pedir todos os documentos que contêm as palavras-chave "motocicleta *and* manutenção", ou documentos que contêm as palavras-chave "computador *or* microprocessador", ou ainda documentos que contêm a palavra-chave "computador *but not* database". Uma consulta contendo palavras-chave sem qualquer um desses conectivos é considerada como tendo *and*s conectando as palavras-chave implicitamente.

Na recuperação de **texto completo**, todas as palavras em cada documento são consideradas como sendo palavras-chave. Para documentos não estruturados, a recuperação de texto completo é essencial, pois pode não haver informação sobre quais palavras no documento são palavras-chave. Usaremos a palavra **termo** para nos referir às palavras em um documento, pois todas as palavras são palavras-chave.

Em sua forma mais simples, um sistema de recuperação de informações localiza e retorna todos os documentos que contêm todas as palavras-chave na consulta, se a consulta não tiver conectivos; os conectivos são tratados como você poderia esperar. Sistemas mais sofisticados estimam a relevância dos documentos a uma consulta de modo que os documentos possam ser mostrados na ordem de relevância estimada. Eles usam informações sobre ocorrências de termo, além de informações de hiperlink, para estimar a relevância.

Classificação de relevância usando termos

O conjunto de todos os documentos que satisfazem uma expressão de consulta pode ser muito grande; em particular, existem bilhões de documentos na Web, e a maioria das consultas de palavra-chave em um mecanismo de busca da Web encontra centenas de milhares de documentos contendo as palavras-chave. A recuperação de texto completo piora esse problema: cada documento pode conter muitos termos, e até mesmo termos que são apenas mencionados de passagem são tratados de forma equivalente a documentos em que o termo é realmente relevante. Os documentos irrelevantes podem ser apanhados como resultado.

Os sistemas de recuperação de informações, portanto, estimam a relevância de documentos a uma consulta e retornam apenas documentos com avaliação alta como respostas. A classificação da relevância não é uma ciência exata, mas existem algumas técnicas bem aceitas.

Classificação usando TF-IDF

A primeira questão a considerar é, dado um termo t em particular, qual é a relevância de um documento em particular *d* ao termo. Uma técnica é usar o número de ocorrências do termo no documento como uma medida de sua relevância, supondo que os termos relevantes provavelmente serão mencionados muitas vezes em um documento. Apenas contando o número de ocorrências de um termo normalmente não é um bom indicador: primeiro, o número de ocorrências depende do tamanho do documento, e segundo, um documento contendo 10 ocorrências de um termo pode não ser 10 vezes mais relevante do que um documento contendo uma ocorrência.

Uma forma de medir $TF(d, t)$, a relevância de um documento d a um termo t, é

$$TF(d, t) = \log\left(1 + \frac{n(d,t)}{n(d)}\right)$$

onde $n(d)$ indica o número de termos no documento e $n(d, t)$ indica o número de ocorrências do termo t no documento d. Observe que essa medida leva em consideração a extensão do documento. A relevância aumenta com mais ocorrências de um termo no documento, embora não seja diretamente proporcional ao número de ocorrências.

Muitos sistemas refinam essa métrica usando outras informações. Por exemplo, se o termo ocorre no título, ou na lista de autores, ou no resumo, o documento seria considerado mais relevante ao termo. De modo semelhante, se a primeira ocorrência de um termo estiver muito longe do início do documento, este pode ser considerado menos relevante do que se a primeira ocorrência estivesse no início do documento. Essas noções podem ser formalizadas

por extensões da fórmula que mostramos para $TF(d, t)$. Na comunidade de recuperação de informações, a relevância de um documento a um termo é conhecida como **freqüência de termo** (FT), independente da fórmula exata sendo utilizada.

Uma consulta Q pode conter várias palavras-chave. A relevância de um documento a uma consulta com duas ou mais palavras-chave é estimada pela combinação das medidas de relevância do documento a cada palavra-chave. Um modo simples de combinar as medidas é acumulá-las. Porém, nem todos os termos usados como palavras-chave são iguais. Suponha que uma consulta use dois termos, um dos quais ocorre com freqüência, como "database", e outro que é menos freqüente, como "Silberschatz". Um documento contendo "Silberschatz", mas não "database", deverá ter uma classificação mais alta do que um documento contendo o termo "database", mas não "Silberschatz".

Para resolver esse problema, pesos são atribuídos aos termos usando a **freqüência de documento inversa** (IDF – Inverse Document Frequency), definida como

$$IDF(t) = \frac{1}{n(t)}$$

onde $n(t)$ indica o número de documentos (entre aqueles indexados pelo sistema) que contêm o termo t. A **relevância** de um documento d a um conjunto de termos Q é, então, definida como

$$r(d, Q) = \sum_{t \in Q} TF(d, t) * IDF(t)$$

Essa medida pode ser refinada ainda mais se o usuário tiver permissão para especificar pesos $w(t)$ para os termos na consulta, quando os pesos especificados pelo usuário também são levados em consideração multiplicando-se $TF(t)$ por $w(t)$ na fórmula anterior.

Essa técnica de usar freqüência de termo e freqüência de documento inversa como medida da relevância de um documento é denominada técnica **TFIDF**.

Quase todos os documentos de texto (em inglês) contêm palavras como "and", "or", "a", e assim por diante, e por isso essas palavras são inúteis para fins de consulta, pois sua freqüência de documento inversa é extremamente baixa. Os sistemas de recuperação de informações definem um conjunto de palavras, chamadas **palavras de parada**, contendo 100 ou mais das palavras mais comuns, e ignoram essas palavras ao indexar um documento. Essas palavras não são usadas como palavras-chave, e são descartadas se estiverem presentes nas palavras-chave fornecidas pelo usuário.

Outro fator levado em consideração quando uma consulta contém vários termos é a **proximidade** dos termos no documento. Se os termos ocorrem perto um do outro no documento, o documento teria uma classificação mais alta do que se ocorrerem afastados. A fórmula para $r(d, Q)$ pode ser modificada para levar em conta a proximidade dos termos.

Dada uma consulta Q, a tarefa de um sistema de recuperação de informações é retornar documentos em ordem decrescente de sua relevância a Q. Como pode haver uma grande quantidade de documentos relevantes, os sistemas de recuperação de informações normalmente retornam apenas os primeiros documentos com o mais alto grau de relevância estimada, e permitem que os usuários solicitem outros documentos interativamente.

Recuperação baseada em semelhança

Certos sistemas de recuperação de informações permitem a **recuperação baseada em semelhança**. Aqui, o usuário pode dar o documento do sistema A e pedir que o sistema apanhe documentos que são "semelhantes" a A. A semelhança de um documento com outro pode ser definida, por exemplo, com base nos termos comuns. Uma técnica é encontrar k termos em A com valores mais altos de $TF(A,t) * IDF(t)$, e usar esses k termos como uma consulta para encontrar a relevância de outros documentos. Os termos na consulta são pesados por $TF(A, t) * IDF(t)$.

Geralmente, a semelhança de documentos é definida pela métrica de **semelhança do cosseno**. Considere que os termos ocorrendo em qualquer um dos dois documentos sejam $t_1, t_2, \ldots, t_n$. Considere que $r(d,t) = TF(d,t) * IDF(t)$. Então, a métrica de semelhança do cosseno entre os documentos d e e é definida como

$$\frac{\sum_{i=1}^{n} r(d, t_i) r(e, t_i)}{\sqrt{\sum_{i=1}^{n} r(d, t_i)^2} \sqrt{\sum_{i=1}^{n} r(e, t_i)^2}}$$

Facilmente se pode verificar que a métrica de semelhança do cosseno de um documento consigo mesmo é 1, enquanto entre dois documentos que não compartilham termo algum é 0.

O nome "semelhança do cosseno" vem do fato de que a fórmula apresentada calcula o cosseno do ângulo entre dois vetores, cada um representando um documento, definido da seguinte maneira. Considere que existem n palavras em geral por todos os documentos sendo considerados. Um espaço de n dimensões é definido, com cada palavras como uma das dimensões. Um documento d é representado por um ponto nesse espaço, como valor da i-ésima coordenada do ponto sendo $r(d, t_i)$. O vetor para o documento d conecta a origem (todas as coordenadas = 0) ao ponto representando o documento. O modelo dos documentos como pontos e vetores em um espaço de n dimensões é chamado **modelo de espaço de vetor**.

Se o conjunto de documentos semelhantes a um documento de consulta A for grande, o sistema pode apresentar

ao usuário alguns dos documentos semelhantes, permitir que o usuário escolha os mais relevantes e iniciar uma nova busca com base na semelhança com *A e* com os documentos escolhidos. O conjunto de documentos resultante provavelmente será o que o usuário desejava encontrar. Essa idéia é chamada **feedback por relevância**.

O feedback por relevância também pode ser usado para ajudar os usuários a encontrar documentos relevantes a partir de um grande conjunto de documentos combinando com as palavras-chave de consulta indicadas. Nessa situação, os usuários podem ter permissão para identificar um ou alguns dos documentos retornados como relevantes; o sistema, então, usa os documentos identificados para encontrar outros semelhantes. O conjunto de documentos resultante provavelmente será o que o usuário desejava encontrar. Uma alternativa à técnica de feedback por relevância é exigir que os usuários modifiquem a consulta acrescentando mais palavras-chave; o feedback por relevância pode ser mais fácil de usar, além de oferecer um melhor conjunto final de documentos como resposta.

Para mostrar ao usuário um conjunto representativo de documentos quando o número de documentos é muito grande, um sistema de busca pode agrupar os documentos, com base na semelhança do cosseno. O agrupamento foi descrito anteriormente na seção "Agrupamento" do Capítulo 18, e várias técnicas foram desenvolvidas para agrupar conjuntos de documentos. As notas bibliográficas possuem referências a mais informações sobre agrupamento.

Relevância usando hiperlinks

Os primeiros mecanismos de busca da Web classificavam os documentos usando apenas medidas de relevância baseadas em TFIDF, como aquelas descritas na seção "Classificação de relevância usando termos". Porém, essas técnicas tinham algumas limitações quando usadas em coleções muito grandes de documentos, como o conjunto de todas as páginas Web. Em particular, muitas páginas Web têm todas as palavras-chave especificadas em uma consulta típica do mecanismo de busca; além do mais, algumas das páginas que os usuários desejam como resposta normalmente possuem apenas algumas ocorrências dos termos da consulta e não receberiam uma avaliação TFIDF muito alta.

Porém, os pesquisadores logo observaram que as páginas Web têm informações muito importantes que os documentos de texto puro não têm, ou seja, hiperlinks. Estes podem ser explorados para a melhor avaliação da relevância; em particular, a classificação por relevância de uma página é bastante influenciada por hiperlinks que apontam *para* a página. Nesta seção, estudamos como os hiperlinks são usados para a classificação de páginas Web.

Classificação por popularidade

A idéia básica da **classificação** por popularidade (também chamada **classificação por prestígio**) é encontrar páginas que são populares e classificá-las antes de outras páginas que contêm as palavras-chave especificadas. Como a maioria das buscas pretende encontrar informações de páginas populares, a classificação dessas páginas na frente geralmente é uma boa idéia. Por exemplo, o termo "google" pode ocorrer em muitas páginas, mas a página google.com é a mais popular entre aquelas que contêm o termo "google". A página google.com, portanto, deve ser classificada como a resposta mais relevante a uma consulta consistindo no termo "google".

As medidas tradicionais de relevância de uma página, como as medidas baseadas em TFIDF, que vimos na seção "Classificação de relevância usando termos", podem ser combinadas com a popularidade da página, para obter uma medida geral da relevância da página à consulta. Páginas com o mais alto valor de relevância geral são retornadas como primeiras respostas a uma consulta.

Isso levanta a questão de como definir e como encontrar a popularidade de uma página. Um modo seria descobrir quantas vezes uma página é acessada e usar o número como uma medida da popularidade dos sites. Porém, é impossível obter essa informação sem a cooperação do site, cuja implementação é inviável para um mecanismo de pesquisa na Web.

Uma alternativa bastante eficaz é usar hiperlinks para uma página como medida de sua popularidade. Muitas pessoas possuem arquivos de marcador que contêm links para sites que usam com freqüência. Os sites que aparecem em uma grande quantidade de arquivos de marcador podem ser deduzidos como sendo sites muito populares. Os arquivos de marcador normalmente são armazenados de forma privada, e não são acessíveis na Web. Porém, muitos usuários mantêm páginas Web como links para suas páginas Web favoritas. Muitos sites também possuem links para outros sites relacionados, o que também pode ser usado para deduzir a popularidade dos sites vinculados. Um mecanismo de busca na Web pode apanhar páginas Web (por um processo chamado vasculhada, que descrevemos na seção "Mecanismos de busca na Web") e analisá-las para encontrar links entre as páginas.

Uma primeira solução para estimar a popularidade de uma página é usar o número de páginas que se vinculam à página como uma medida de sua popularidade. Porém, isso por si só tem a desvantagem de que muitos sites têm uma série de páginas úteis, enquanto links externos normalmente apontam somente para a página raiz do site. A página raiz, por sua vez, possui links para outras páginas no site. Essas outras páginas, então, seriam indevidamente deduzidas como não sendo muito populares, e teriam uma avaliação baixa na resposta às consultas.

Uma alternativa é associar a popularidade aos sites, em vez das páginas. Todas as páginas em um site, então, recebem a popularidade do site, e as páginas diferentes da página raiz de um site popular também se beneficiariam com a popularidade dos sites. Porém, surge a questão quanto ao que constitui um site. Em geral, o prefixo do endereço da Internet de um URL de página constituiria o site correspondente à página. Porém, existem muitos sites que hospedam uma grande quantidade de páginas principalmente não relacionadas, como servidores de home page em universidades e portais Web, como groups.yahoo.com ou tripod.com. Para esses sites, a popularidade de uma parte do site não implica na popularidade de outra parte do site.

Uma alternativa mais simples é permitir a *transferência de prestígio* de páginas populares para as páginas às quais elas estão vinculadas. Sob esse esquema, ao contrário dos princípios de "uma pessoa, um voto" da democracia, um link de uma página popular x para uma página y é tratado como possuindo mais prestígio para a página y do que um link de uma página não tão popular z.[1]

Essa noção de popularidade, na verdade, é circular, pois a popularidade de uma página é definida pela popularidade de outras páginas, e pode haver ciclos de links entre as páginas. Porém, a popularidade das páginas pode ser definida por um sistema de equações lineares simultâneas, que podem ser resolvidas por técnicas de manipulador de matriz. As equações lineares podem ser definidas de modo que tenham uma solução exclusiva e bem definida.

É interessante observar que a idéia básica por trás da classificação básica de popularidade é realmente muito antiga, e apareceu inicialmente em uma teoria de rede social desenvolvida pelos sociólogos na década de 1950. No contexto das redes sociais, o objetivo era definir o prestígio das pessoas. Por exemplo, o presidente dos Estados Unidos tem alto prestígio, pois um grande número de pessoas o conhece. Se alguém é conhecido por várias pessoas de prestígio, então ele tem alto prestígio, mesmo que não seja conhecido por um grande número de pessoas. O uso de um conjunto de equações lineares para definir a medida da popularidade também vem desde a época desse trabalho.

PageRank

O mecanismo de busca da Web Google introduziu o **PageRank**, que é uma medida da popularidade de uma página baseada na popularidade das páginas que se vinculam à página. O uso da medida de popularidade PageRank para classificar as respostas a uma consulta deu resultados tão melhores do que as técnicas de classificação usadas anteriormente que o Google se tornou o mecanismo de busca mais utilizado, em um período de tempo bem curto.

PageRank pode ser entendido intuitivamente por meio de um **modelo de caminhada aleatória**. Suponha que uma pessoa navegando a Web realize uma caminhada aleatória (travessia) nas páginas Web da seguinte maneira: o primeiro passo começa em uma página Web aleatória e, a cada passo, o caminhante aleatório faz um dos seguintes. Com uma probabilidade δ, ele salta para uma página Web escolhida aleatoriamente, e com uma probabilidade de $1-\delta$, ele escolhe aleatoriamente um dos links externos da página Web atual e segue o link. O PageRank de uma página, então, é a probabilidade de que o caminhante aleatório esteja visitando a página em algum ponto qualquer no tempo.

Observe que as páginas que são apontadas a partir de muitas páginas Web provavelmente serão mais visitadas, e por isso terão um PageRank mais alto. De modo semelhante, as páginas apontadas por páginas Web com um PageRank alto também terão uma probabilidade maior de serem visitadas, e por isso terão um PageRank mais alto.

PageRank pode ser definido por um conjunto de equações lineares, como a seguir. Primeiro, as páginas Web recebem identificadores inteiros. A matriz de probabilidade de salto T é definida com $T[i,j]$ definido como a probabilidade de que um caminhante aleatório que esteja seguindo um link para fora da página i siga o link para a página j. Supondo que cada link de i tenha uma probabilidade igual de ser seguido, $T[i,j] = 1/N_i$, onde N_i é o número de links para fora da página i. A maioria das entradas de T é 0 e é melhor representada como uma lista de adjacência. Então, o PageRank $P[j]$ para cada página j pode ser definido como

$$P[j] = \delta/N + (1-\delta) * \sum_{i=1}^{N} (T[i,j] * P[i])$$

onde δ é uma constante entre 0 e 1, e N é o número de páginas; δ representa a probabilidade de uma etapa na caminhada aleatória ser um salto.

O conjunto dessas equações geradas normalmente é solucionado por uma técnica iterativa, começando com cada $P[i]$ definido como $1/N$. Cada etapa da iteração calcula novos valores para cada $P[i]$ usando os valores P da iteração anterior. A iteração termina quando a mudança máxima em qualquer valor $P[i]$ em uma iteração fica abaixo de algum valor de limite.

Outras medidas de popularidade

As medidas básicas de popularidade, como PageRank, desempenham um papel importante na classificação das respostas à consulta, mas de forma alguma são o único fator.

1. De certa forma, isso é semelhante ao peso extra dado aos endossos de produtos por celebridades (como estrelas de filmes), de modo que seu significado é aberto a questionamentos, embora sendo eficiente e bastante usado na prática.

As notas TFIDF de uma página são usadas para julgar sua relevância às palavras-chave da consulta, e precisam ser combinadas com a classificação da popularidade. Outros fatores também precisam ser levados em conta, para lidar com limitações de PageRank e medidas de popularidade relacionadas.

Uma desvantagem do algoritmo PageRank é que ele atribui uma medida da popularidade que não leva em conta as palavras-chave da consulta. Por exemplo, a página `google.com` provavelmente terá um PageRank muito alto, pois muitos sites contêm um link para ela. Suponha que ela contenha uma palavra mencionada de passagem, como "Stanford" (a página de busca avançada no Google de fato continha essa palavra, pelo menos no início de 2005). Uma busca sobre a palavra-chave Stanford então retornaria `google.com` como a resposta com classificação mais alta, antes de uma reposta mais relevante, como a página Web da Universidade de Stanford.

Uma solução bastante usada para esse problema é usar palavras-chave no texto de âncora dos links para uma página, a fim de julgar a que tópicos a página é altamente relevante. O texto de âncora de um link consiste no texto que aparece com a tag HTML a href. Por exemplo, o texto de âncora do link

```
<a href="http://stanford.edu"> Stanford University</a>
```

é "Stanford University". Se muitos links para stanford.edu tiverem a palavra Stanford em seu texto de âncora, o site pode ser considerado muito relevante à palavra-chave Stanford. O texto próximo do texto de âncora também pode ser levado em conta; por exemplo, um site pode conter o texto "a página principal de Stanford está aqui", mas pode ter usado apenas a palavra "aqui" como texto de âncora no link para o site de Stanford.

A popularidade baseada no texto de âncora é combinada com outras medidas de popularidade, e com as medidas TFIDF, para chegar a uma classificação geral para repostas à consulta. Observamos que a maioria dos mecanismos de busca não revela como eles calculam as classificações de relevância; eles acreditam que revelar suas técnicas de classificação daria vantagens aos concorrentes e facilitaria o trabalho de "spamming de mecanismo de busca", resultando na qualidade inferior dos resultados. O spamming de mecanismo de busca é descrito com mais detalhes em outra parte desta seção.

Uma técnica alternativa para levar em conta as palavras-chave ao definir a popularidade é calcular uma medida de popularidade usando *apenas* páginas que contêm as palavras-chave da consulta, em vez de calcular a popularidade usando todas as páginas Web disponíveis. Essa técnica é mais dispendiosa, pois o cálculo da classificação de popularidade precisa ser feito dinamicamente, quando uma consulta é recebida, enquanto o PageRank é calculado estaticamente uma vez, e reutilizado para todas as consultas. Os mecanismos de busca da Web tratando de bilhões de consultas por dia não podem gastar tanto tempo respondendo a uma consulta. Como resultado, embora essa técnica possa dar melhores respostas, ela não é muito utilizada.

O algoritmo HITS foi baseado nessa idéia de encontrar primeiro as páginas que contêm as palavras-chave da consulta e depois calcular uma medida de popularidade usando apenas esse conjunto de páginas relacionadas. Além disso, ele introduziu uma noção de *hubs* e *autoridades*. Um **hub** é uma página que armazena links para muitas páginas relacionadas; ele não pode por si só conter informações reais sobre um assunto, mas aponta para páginas que contêm informações reais. Ao contrário, uma **autoridade** é uma página que contém informações reais sobre um assunto, embora possa não armazenar links para muitas páginas relacionadas. Cada página, então, recebe um valor de prestígio como um hub (*prestígio de hub*), e outro valor de prestígio como uma autoridade (*prestígio de autoridade*). As definições de prestígio, como antes, são cíclicas e definidas por um conjunto de equações lineares simultâneas. Uma página obtém prestígio de hub mais alto se apontar para muitas páginas com alto prestígio de autoridade, enquanto uma página recebe prestígio de autoridade mais alto se for apontada por muitas páginas com alto prestígio de hub. Em uma consulta qualquer, as páginas com prestígio de autoridade mais alto têm uma avaliação mais alta do que outras páginas. Veja mais detalhes sobre isso nas referências contidas nas notas bibliográficas deste capítulo.

O **spamming do mecanismo de busca** refere-se à prática de criar páginas Web, ou conjuntos de páginas Web, projetadas para receber uma alta classificação de relevância para algumas consultas, embora os sites não sejam realmente populares. Por exemplo, um site de viagens pode querer ter uma avaliação alta para consultas com a palavra-chave "viagem". Ele pode receber notas TFIDF altas repetindo a palavra "viagem" muitas vezes em sua página.[2] Até mesmo um site não relacionado com a viagem, como um site pornográfico, poderia fazer o mesmo e receberia uma classificação bem alta para uma consulta sobre a palavra "viagem". De fato, esse tipo de spamming de TFIDF era comum nos primeiros dias da busca na Web, e havia uma batalha constante entre esses sites e os mecanismos de busca, que tentavam detectar o spamming e negar-lhes uma classificação alta.

2. Palavras repetidas em uma página Web podem confundir os usuários, os spammers podem enfrentar esse problema oferecendo diferentes páginas aos mecanismos de busca do que outros usuários, para o mesmo URL, ou tornando as palavras repetidas invisíveis, por exemplo, formatando-as em uma fonte branca com corpo pequeno contra um fundo branco.

Esquemas de classificação de popularidade como PageRank dificultam o trabalho de spamming do mecanismo de busca, pois apenas repetir palavras para conseguir uma avaliação TFIDF alta não é mais suficiente. Porém, até mesmo essas técnicas podem ser burladas, criando uma coleção de páginas Web que apontam uma para a outra, aumentando a classificação da popularidade. Técnicas como usar sites no lugar de páginas como a unidade de classificação (com probabilidades de salto devidamente normalizadas) têm sido propostas para evitar algumas técnicas de spamming, mas não são totalmente eficazes contra outras técnicas de spamming. A guerra entre os spammers dos mecanismos de busca e os mecanismos de busca continua ainda hoje.

A técnica de hubs e autoridades do algoritmo HITS é mais suscetível ao spamming. Um spammer pode criar uma página Web contendo links para boas autoridades sobre um assunto e, como resultado, ganhar uma alta classificação de hub. Além disso, a página Web dos spammers inclui links para as páginas que eles querem popularizar, que podem não ter qualquer relevância ao assunto. Como essas páginas Web vinculadas são apontadas por uma página com uma alta avaliação de hub, elas recebem uma nota de autoridade alta, porém, imerecida.

Sinônimos, homônimos e ontologias

Considere o problema de localizar documentos sobre manutenção de motocicleta, usando a consulta "manutenção motocicleta". Suponha que as palavras-chave para cada documento sejam as palavras no título e os nomes dos autores. O documento intitulado *Reparo Motocicleta* não seria apanhado, pois a palavra "manutenção" não ocorre em seu título.

Podemos resolver esse problema utilizando **sinônimos**. Cada palavra pode ter um conjunto de sinônimos definidos, e a ocorrência de uma palavra pode ser substituída pelo *or* de todos os seus sinônimos (incluindo a própria palavra). Assim, a consulta "motocicleta *and* reparo" pode ser substituída por "motocicleta *and* (reparo *or* manutenção)". Essa consulta encontraria o documento desejado.

As consultas baseadas em palavra-chave também sofrem com o problema oposto, o de **homônimos**, ou seja, palavras isoladas com vários significados. Por exemplo, a palavra "guarda" tem significados diferentes como um nome e como um verbo. A palavra "mesa" pode se referir a uma mesa de jantar, ou a uma mesa de apoio de uma empresa.

De fato, um perigo mesmo com o uso de sinônimos para estender consultas é que os sinônimos por si só possam ter significados diferentes. Por exemplo, "suporte" é um sinônimo para um significado da palavra "manutenção", mas tem um significado diferente daquele que o usuário pensou na formulação da consulta "manutenção motocicleta". Os documentos que utilizam os sinônimos com um significado alternativo intencionado também seriam apanhados. O usuário, então, fica sem saber por que o sistema pensou que determinado documento apanhado (por exemplo, usando a palavra "suporte") é relevante, se ele não contém nem as palavras que o usuário especificou, nem palavras cujo significado intencionado no documento seja sinônimo das palavras-chave especificadas! Portanto, é uma má idéia usar sinônimos para estender uma consulta sem primeiro verificar os sinônimos com o usuário.

Uma técnica melhor para esse problema é que o sistema entenda que *conceito* cada palavra em um documento representa e, de modo semelhante, entenda que conceitos um usuário está procurando, para retornar documentos que consideram os conceitos em que o usuário está interessado. Um sistema que admite a **consulta baseada em conceito** precisa analisar cada documento para retirar a ambigüidade de cada palavra no documento, e substituí-la pelo conceito que ela representa; a retirada da ambigüidade normalmente é feita procurando-se outras palavras ao redor no documento. Por exemplo, se um documento contém palavras como apoio ou suporte, a palavra "mesa" provavelmente deve ser substituída pelo conceito "mesa: apoio", enquanto, se o documento contém palavras como móveis, cadeia ou madeira próximas da palavra mesa, a palavra deve ser substituída pelo conceito "mesa: móvel". A retirada da ambigüidade baseada em palavras vizinhas normalmente é mais difícil para as consultas do usuário, pois elas contêm muito poucas palavras, de modo que os sistemas de consulta baseada em conceito ofereceriam vários conceitos alternativos ao usuário, que escolhe um ou mais antes que a busca continue.

A consulta baseada em conceito tem diversas vantagens; por exemplo, uma consulta em uma linguagem pode apanhar documentos em outras linguagens, desde que se relacionem ao mesmo conceito. Mecanismos de tradução automatizados podem ser usados mais tarde se o usuário não entender a linguagem em que o documento foi escrito. Porém, a sobrecarga do processamento de documentos para retirar a ambigüidade das palavras é muito alta quando trata de bilhões de documentos. Os mecanismos de busca da Internet, portanto, geralmente não admitem a consulta baseada em conceito. Porém, os sistemas de consulta baseada em conceito foram criados e usados para outras grandes coleções de documentos.

A consulta baseada em conceitos pode ser estendida ainda mais explorando as hierarquias de conceitos. Por exemplo, suponha que uma pessoa emita uma consulta "animais voadores"; um documento contendo informações sobre "mamíferos voadores" certamente é relevante, pois um mamífero é um animal. Porém, os dois conceitos não são iguais, e apenas combinar conceitos não permitiria que o

documento fosse retornado como resposta. Os sistemas de consulta baseados em conceito podem aceitar a recuperação de documentos com base em hierarquias de conceitos.

Ontologias são estruturas hierárquicas que refletem relacionamentos entre conceitos. O relacionamento mais comum é um relacionamento é **um**; por exemplo, um leopardo é um mamífero, e um mamífero é um animal. Outros relacionamentos, como *parte-de* também são possíveis; por exemplo, uma asa de avião é parte de um avião.

O sistema WordNet define uma grande variedade de conceitos com palavras associadas (chamadas *synset* na terminologia do WordNet). As palavras associadas a um synset são sinônimas para o conceito; certamente, uma palavra pode ser um sinônimo para vários conceitos diferentes. Além de sinônimos, o WordNet define homônimos e outros relacionamentos. Em particular, os relacionamentos "é um" e "parte de" que ele define conecta conceitos e, com efeito, define uma ontologia. O projeto Cyc foi outro esforço para criar uma ontologia.

Além das ontologias em nível de linguagem, outras ontologias foram definidas para áreas específicas, de modo a lidar com a terminologia relevante a essa área. Por exemplo, as ontologias foram criadas para padronizar termos usados nos negócios; esse é um passo importante na criação de uma infra-estrutura padrão para lidar com processamento de pedidos e outros fluxos de dados dentro da organização.

É possível criar ontologias que vinculam vários idiomas. Por exemplo, WordNets foram criadas para diferentes idiomas, e conceitos comuns entre eles podem ser vinculados um ao outro. Tal sistema pode ser usado para a tradução de texto. No contexto da recuperação de informações, uma ontologia de múltiplos idiomas pode ser usada para implementar a busca baseada em conceito entre documentos em vários idiomas.

Indexação de documentos

Uma estrutura de índice eficiente é importante para o processamento eficaz de consultas em um sistema de recuperação de informações. Os documentos que contêm uma palavra-chave especificada podem ser localizados de forma eficiente pelo uso de um **índice invertido**, que mapeia cada palavra-chave K_i a uma lista S_i de (identificadores de) documentos que contêm K_i. Para dar suporte à classificação de relevância baseada na proximidade de palavras-chave, tal índice pode oferecer não apenas identificadores de documentos, mas também uma lista de locais dentro do documento em que a palavra-chave aparece. Esses índices precisam ser armazenados em disco, e cada lista S_i pode se espalhar por várias páginas de disco. A fim de minimizar o número de operações de E/S para apanhar cada lista S_i, o sistema tentaria manter cada lista S_i em um conjunto de páginas de disco consecutivas, de modo que a lista inteira possa ser apanhada com apenas uma busca de disco. Um índice de árvore B+ pode ser usado para mapear cada palavra-chave K_i à sua lista invertida associada, S_i.

A operação *and* encontra documentos que contêm todas de um conjunto de palavras-chave $K_1, K_2, \ldots, K_n$. Implementamos a operação *and* primeiro apanhando os conjuntos de identificadores de documento $S_1, S_2, \ldots, S_n$ de todos os documentos que contêm as respectivas palavras-chave. A interseção dos conjuntos, $S_1 \cap S_2 \cap \ldots \cap S_n$, gera o conjunto de todos os documentos que contêm pelo menos uma das palavras-chave $K_1, K_2, \ldots, K_n$. Implementamos a operação *or* calculando a união dos conjuntos, $S_1 \cup S_2 \cup \ldots \cup S_n$. A operação *not* encontra documentos que não contêm uma palavra-chave K especificada. Dado um conjunto de identificadores de documento S, podemos eliminar documentos que contêm a palavra-chave especificada K_i, apanhando a diferença $S - S_i$, onde S_i é o conjunto de identificadores de documentos que contêm a palavra-chave K_i.

Dado um conjunto de palavras-chave em uma consulta, muitos sistemas de recuperação de informações não requerem que os documentos apanhados contenham todas as palavras-chave (a menos que uma operação *and* seja usada explicitamente). Nesse caso, todos os documentos contendo pelo menos uma das palavras são recuperados (como na operação *or*), mas são classificados por sua medida de relevância.

Para usar a freqüência do termo na classificação, a estrutura de índice deverá, além disso, manter o número de vezes que os termos ocorrem em cada documento. Para reduzir esse esforço, eles podem usar uma representação compactada com apenas alguns bits que se aproximam do termo freqüência. O índice também deve armazenar a freqüência de documentos para termo (ou seja, o número de documentos em que o termo aparece).

A lista S_i pode ser armazenada por classificação de popularidade (e, secundariamente, por documentos com a mesma classificação de popularidade, sobre $id_{documento}$). Então, uma mesclagem simples pode ser usada para calcular operações *and* e *or*. Os resultados com maior classificação de popularidade apareceriam perto da frente das listas. Para o caso da operação *and*, se ignorarmos a contribuição TFIDF para a nota de relevância, e simplesmente exigirmos que o documento deve conter determinadas palavras-chave, a mesclagem pode parar quando K respostas tiverem sido obtidas, se o usuário exigir apenas as primeiras K respostas.

Medindo a eficácia da recuperação

Cada palavra-chave pode estar contida em uma grande quantidade de documentos; logo, uma representação compacta é crítica para diminuir o uso de espaço do índice.

Assim, os conjuntos de documentos para uma palavra-chave são mantidos em um formato compactado. Para que o espaço de armazenamento seja poupado, o índice às vezes é armazenado de modo que a recuperação seja aproximada; alguns documentos relevantes podem não ser apanhados (chamado de **falsa perda** ou **falso negativo**), ou alguns documentos irrelevantes podem ser apanhados (chamado **falso positivo**). Uma boa estrutura de índice não terá *quaisquer* falsas perdas, mas pode permitir alguns falsos positivos; o sistema pode filtrá-los mais tarde, examinando as palavras-chave que eles realmente contêm. Na indexação da Web, os falsos positivos também não são desejáveis, pois o documento real pode não estar acessível rapidamente para filtragem.

Duas métricas são usadas para medir como um sistema de recuperação de informações é capaz de responder a consultas. A primeira, a **precisão**, mede qual porcentagem dos documentos apanhados é realmente relevante à consulta. A segunda, a **rechamada**, mede que porcentagem dos documentos relevantes à consulta foram apanhados. O ideal de ambos deve ser de 100%.

Precisão e rechamada também são medidas importantes para entender como funciona determinada estratégia de classificação de documento. A estratégia de classificação pode resultar em falsos negativos e falsos positivos, mas em um sentido mais sutil.

- Falsos negativos podem ocorrer quando os documentos são classificados como resultado de documentos relevantes recebendo uma classificação baixa. Se o sistema apanhasse todos os documentos até aqueles com classificação muito baixa, haveria muito poucos falsos negativos. Porém, os humanos raramente olham além das primeiras dezenas de documentos retornados, e assim podem perder documentos relevantes, pois não têm uma boa classificação. Exatamente o que é um falso negativo depende de quantos documentos são examinados. Portanto, em vez de ter um único número como medida de rechamada, podemos medir a rechamada como uma função do número de documentos apanhados.
- Falsos positivos podem ocorrer porque documentos irrelevantes recebem classificações mais altas do que os documentos relevantes. Isso também depende de quantos documentos são examinados. Uma opção é medir a precisão como uma função do número de documentos apanhados.

Uma alternativa melhor e mais intuitiva para medir a precisão é medi-la como uma função da rechamada. Com essa medida combinada, a precisão e a rechamada podem ser calculadas como uma função do número de documentos, se for necessário.

Por exemplo, podemos dizer que, com uma rechamada de 50%, a precisão foi de 75%, enquanto em uma rechamada de 75% a precisão caiu para 60%. Em geral, podemos desenhar um gráfico relacionando precisão com rechamada. Essas medidas podem ser calculadas para consultas individuais, para depois ser calculada a média por um conjunto de consultas em um benchmark de consulta.

Ainda outro problema com a medida da precisão e rechamada está em como definir quais documentos são realmente relevantes e quais não são. De fato, é preciso conhecimento da linguagem natural, e conhecimento da intenção da consulta, para decidir se um documento é relevante ou não. Os pesquisadores, portanto, criaram coleções de documentos e consultas, e marcaram os documentos manualmente como relevantes ou irrelevantes às consultas. Diferentes esquemas de classificação podem ser executados sobre essas coleções, para medir sua precisão e rechamada médias por várias consultas.

Mecanismos de busca na Web

Vasculhadores da Web (Web crawlers) são programas que localizam e reúnem informações na Web. Eles seguem recursivamente os hiperlinks presentes em documentos conhecidos para localizar outros documentos. Um vasculhador apanha os documentos e acrescenta informações encontradas nos documentos a um índice combinado; o documento geralmente não é armazenado, embora alguns mecanismos de busca realizem o cache de uma cópia do documento para dar aos clientes um acesso mais rápido aos documentos.

Como a quantidade de documentos na Web é muito grande, não é possível alcançar a Web inteira em um curto período de tempo; de fato, todos os mecanismos de busca cobrem apenas algumas partes da Web, e não ela inteira, e seus vasculhadores podem levar semanas ou meses para realizar uma única vasculhada por todas as páginas que eles abrangem. Normalmente existem muitos processos executando em várias máquinas envolvidas no trabalho. Um banco de dados armazena um conjunto de links (ou sites) a serem verificados; ele atribui links desse conjunto a cada processo vasculhador. Novos links encontrados durante uma vasculhada são adicionados ao banco de dados e podem ser verificados mais tarde se isso não acontecer imediatamente. As páginas encontradas durante uma passada também são entregues a um sistema de computação e indexação de prestígio, que pode estar sendo executado em uma máquina diferente. As páginas precisam ser novamente apanhadas (ou seja, os links precisam ser novamente seguidos) periodicamente para obter informações atualizadas e descartar sites que não existem mais, de modo que as informações no índice de busca sejam mantidas razoavelmente atualizadas.

Os próprios sistemas de computação e indexação de prestígio são executados em várias máquinas em paralelo. Não é uma boa idéia acrescentar páginas ao mesmo índice que está sendo usado para consultas, pois isso exigiria controle de concorrência sobre o índice e afetaria o desempenho da consulta e da atualização. Em vez disso, uma cópia do índice é usada para responder a consultas enquanto outra cópia é atualizada com páginas recém-verificadas. Em intervalos periódicos, as cópias são trocadas, com a antiga sendo atualizada enquanto a nova cópia está sendo usada para consultas.

Para dar suporte a taxas de consulta muito altas, os índices podem ser mantidos na memória principal, e existem várias máquinas; o sistema direciona seletivamente as consultas para as máquinas balancearem entre elas. Mecanismos de busca populares normalmente possuem dezenas de milhares de máquinas executando as diversas tarefas dos vasculhadores, indexando e respondendo as consultas do usuário.

Recuperação de informações e dados estruturados

Embora os sistemas de recuperação de informações fossem projetados originalmente para encontrar documentos textuais relacionados a uma consulta, existe uma necessidade crescente por sistemas que tentem entender os documentos (de modo limitado) e responder a perguntas com base no conhecimento (limitado). Uma técnica é criar informações estruturadas a partir de documentos não estruturados e responder as perguntas com base na informação estruturada. Outra técnica aplica técnicas de linguagem natural para encontrar documentos relacionados a uma pergunta (em linguagem natural) e retornar segmentos relevantes dos documentos como uma resposta à pergunta.

Extração de informações

Sistemas de **extração de informações** convertem informações do formato textual para um formato mais estruturado. Por exemplo, um anúncio de imóveis pode descrever atributos de uma casa em formato textual, por exemplo "casa de dois quartos e três banheiros em Queens, US$1 milhão", do qual um sistema de extração de informações pode extrair atributos como número de quartos, número de banheiros, custo e local. Essa informação extraída pode ser usada para responder melhor às consultas. Uma organização que mantém um banco de dados de informações da empresa pode usar um sistema de extração de informações para extrair automaticamente as informações dos artigos de jornal; a informação extraída seria relacionada a mudanças nos atributos de interesse, como pedidos de demissão, demissões ou compromissos de dirigentes da empresa. Vários sistemas foram criados para extrair informações para aplicações especializadas. Eles utilizam técnicas linguísticas, bem como regras definidas pelo usuário, para domínios específicos (como anúncios de imóveis).

Consulta de dados estruturados

Os dados estruturados são representados principalmente em formato relacional ou XML. Vários sistemas foram criados para dar suporte à consulta por palavra-chave sobre dados relacionais e XML. Um tema comum entre esses sistemas é a descoberta de nós (tuplas ou elementos XML) contendo as palavras-chave especificadas e a localização de caminhos de conexão (ou ancestrais comuns, no caso de dados XML) entre eles.

Por exemplo, uma consulta por "Smith Queens" sobre um banco de dados bancário poderá encontrar o nome Smith ocorrendo em uma tupla *cliente* e o nome Queens em uma tupla *agência*, além de um caminho pela relação *depositante* conectando as duas tuplas. Essas consultas podem ser usadas para a navegação e consulta ocasional dos dados, quando o usuário não sabe o esquema exato e não deseja ter o trabalho de escrever uma consulta SQL ou XQuery definindo o que está procurando. Em vez disso, é demais esperar que usuários leigos escrevam consultas em uma linguagem de consulta estruturada, enquanto a consulta por palavra-chave é bastante natural.

Como as consultas não são totalmente definidas, elas podem ter muitos tipos diferentes de respostas, que precisam ser classificadas. Diversas técnicas foram propostas para classificar as respostas em tal ambiente, com base nos tamanhos dos caminhos de conexão e em técnicas para atribuir direções e pesos aos caminhos. Também foram propostas técnicas para atribuir níveis de popularidade a tuplas e elementos XML, com base em links como chave estrangeira e links IDREF. Veja, nas notas bibliográficas, mais informações sobre a busca por palavra-chave em dados relacionais e XML.

Resposta a perguntas

Os sistemas de recuperação de informações focalizam a localização de documentos relevantes a determinada consulta. Porém, a resposta a uma consulta pode estar em apenas uma parte de um documento ou em pequenas partes de vários documentos. Os sistemas de **resposta a perguntas** tentam oferecer respostas diretas a perguntas feitas pelos usuários. Por exemplo, uma pergunta no formato "Quem matou Lincoln?" pode ser respondida por uma linha que diz "Abraham Lincoln foi morto a tiros por John Wilkes Booth em 1865". Observe que a resposta não contém realmente as palavras "matou" ou "quem", mas o sistema deduz que

"quem" pode ser respondido por um nome e "matou" está relacionado a "foi morto".

Os sistemas de resposta a perguntas voltados para informações na Web normalmente geram uma ou mais consultas de palavra-chave a partir da pergunta submetida, executam as consultas de palavra-chave contra mecanismos de busca na Web e analisam os documentos retornados para encontrar segmentos dos documentos que respondam à pergunta. Diversas técnicas linguísticas e heurísticas são usadas para gerar consultas de palavra-chave e encontrar segmentos relevantes a partir do documento. O mecanismo de busca MSN da Microsoft admite resposta a perguntas, usando a enciclopédia Encarta como sua principal fonte de informação.

Os sistemas de resposta a perguntas da geração atual são limitados em potência, pois não entendem realmente a pergunta ou os documentos usados para respondê-la. Porém, eles são úteis para diversas tarefas simples de resposta a perguntas.

Diretórios

Um usuário de biblioteca típico pode usar um catálogo para localizar um livro pelo qual está procurando. Contudo, quando ele apanha o livro na prateleira, provavelmente passa por outros livros que estão localizados nas proximidades. As bibliotecas são organizadas de modo que os livros relacionados sejam mantidos próximos. Logo, um livro que está fisicamente perto do livro desejado também pode ser de interesse, tornando o exame de tais livros valioso para os usuários.

Para manter livros relacionados próximos um do outro, as bibliotecas utilizam uma **hierarquia de classificação**. Os livros sobre ciência são classificados juntos. Dentro desse conjunto de livros, existe uma classificação mais detalhada, com os livros de ciência da computação organizados juntos, livros de matemática organizados juntos e assim por diante. Como há uma relação entre matemática e ciência da computação, os conjuntos de livros relevantes são armazenados fisicamente próximos um do outro. Ainda em outro nível na hierarquia de classificação, os livros de ciência da computação são desmembrados em subáreas, como sistemas operacionais, linguagens e algoritmos. A Figura 19.1 ilustra uma hierarquia de classificação que pode ser usada por uma biblioteca. Como os livros só podem ser mantidos em um local, cada livro em uma biblioteca é classificado exatamente em um ponto na hierarquia de classificação.

Em um sistema de recuperação de informações, não é necessário armazenar documentos relacionados próximos um do outro. Porém, esses sistemas precisam *organizar documentos logicamente*, de modo a permitir a navegação. Assim, tal sistema poderia usar uma hierarquia de classificação semelhante àquela que as bibliotecas utilizam e, quando exibe determinado documento, também pode exibir uma rápida descrição dos documentos que estão próximos na hierarquia.

Em um sistema de recuperação de informações, não é necessário manter um documento em um único local na hierarquia. Um documento que fala de matemática para cientistas da computação poderia ser classificado sob matemática e também sob ciência da computação. Tudo o que é armazenado em cada ponto é um identificador do documento (ou seja, um ponteiro para o documento), e é fácil apanhar o conteúdo do documento usando o identificador.

Como resultado dessa flexibilidade, não apenas um documento pode ser classificado sob dois locais, mas também

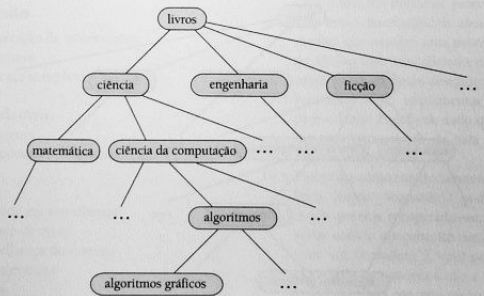


Figura 19.1 Uma hierarquia de classificação para um sistema de biblioteca.

uma própria subárea da hierarquia de classificação pode ocorrer sob duas áreas. A classe de documentos "algoritmos gráficos" pode aparecer sob matemática e sob ciência da computação. Assim, a hierarquia de classificação agora é um grafico acíclico direcionado (DAG – Directed Acyclic Graph), como mostra a Figura 19.2. Um documento sobre algoritmos gráficos pode aparecer em um único local no DAG, mas pode ser alcançado por vários caminhos.

Um **diretório** é simplesmente uma estrutura DAG de classificação. Cada folha do diretório armazena links para documentos sobre o assunto representado pela folha. Os nós internos também podem conter links, por exemplo, para documentos que não podem ser classificados sob qualquer um dos nós filhos.

Para localizar informações sobre um assunto, um usuário começaria na raiz do diretório e seguiria os caminhos pelo DAG até alcançar um nó que represente o assunto desejado. Enquanto navega pelo diretório, o usuário pode descobrir não apenas documentos sobre o assunto em que está interessado, mas também encontrar documentos relacionados e classes relacionadas na hierarquia de classificação. O usuário pode descobrir novas informações navegando pelos documentos (ou subclasses) dentro das classes relacionadas.

A organização da enorme quantidade de informações disponíveis na Web em uma estrutura de diretório é uma tarefa assustadora.

- O primeiro problema é determinar qual exatamente deve ser a hierarquia do diretório.
- O segundo problema é, dado um documento, decidir quais nós do diretório são categorias relevantes ao documento.

Para resolver o primeiro problema, portais como Yahoo possuem equipes de "bibliotecários da Internet" que aparecem com a hierarquia de classificação e a refinam continuamente. O *Open Directory Project* é um grande esforço colaborativo, com diferentes voluntários sendo responsáveis pela organização de diferentes ramos do diretório.

O segundo problema também pode ser resolvido manualmente pelos bibliotecários, ou os mantenedores de site podem ser responsáveis por decidir onde estes devem ficar na hierarquia. Há também técnicas para decidir automaticamente o local dos documentos, com base no cálculo de sua semelhança com documentos que já foram classificados.

Resumo

- Os sistemas de recuperação de informações são usados para armazenar e consultar dados textuais como documentos. Eles utilizam um modelo de dados mais simples do que os sistemas de banco de dados, mas oferecem capacidades de consulta mais poderosas dentro do modelo restrito.

  As consultas tentam localizar documentos que são de interesse especificando, por exemplo, conjuntos de palavras-chave. A consulta que um usuário tem em mente normalmente não pode ser expressa com precisão; logo, os sistemas de recuperação de informações ordenam as respostas com base na relevância em potencial.
- A classificação de relevância utiliza vários tipos de informação, como:
  - Freqüência do termo: qual é a importância do termo para cada documento.
  - Freqüência inversa do documento.
  - Classificação de popularidade.

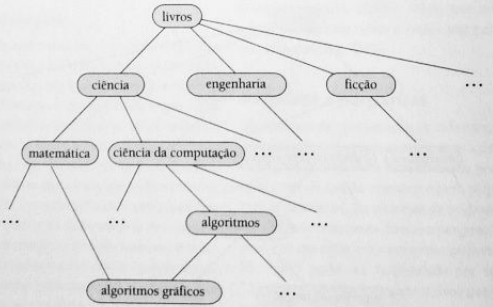


Figura 19.2 Uma classificação DAG para um sistema de recuperação de informações de biblioteca.

- A semelhança dos documentos é usada para apanhar documentos semelhantes a um documento de exemplo. A métrica do cosseno é usada para definir a semelhança, sendo baseada no modelo de espaço de vetor.
- PageRank e a classificação por hub/autoridade são duas maneiras de atribuir prestígio às páginas, com base nos links com essa página. A medida PageRank pode ser entendida intuitivamente usando um modelo de caminhada aleatória. A informação do texto de âncora também é usada para calcular uma noção de popularidade por palavra-chave.
- O spamming do mecanismo de busca tenta conseguir uma classificação alta (não merecida) para uma página.
- Sinônimos e homônimos complicam a tarefa de recuperação de informações. A consulta baseada em conceito procura localizar documentos contendo conceitos especificados, independente das palavras exatas (ou linguagem) em que o conceito é especificado. As ontologias são usadas para relacionar conceitos usando relacionamentos do tipo é-um ou parte-de.
- Os índices invertidos são usados para responder a consultas por palavra-chave.
- Precisão e rechamada são duas medidas da eficácia de um sistema de recuperação de informações.
- Os mecanismos de busca da Web vasculham a Web para encontrar páginas, analisá-las para calcular medidas de prestígio e indexá-las.
- Foram desenvolvidas técnicas para extrair informações estruturadas a partir de dados textuais, para realizar a consulta por palavra-chave sobre dados estruturados e oferecer respostas diretas a perguntas simples, feitas em linguagem natural.
- Estruturas de diretório são usadas para classificar documentos com outros documentos semelhantes.

Termos de revisão

- Sistemas de recuperação de informações
- Busca de palavra-chave
- Recuperação de texto completo
- Termo
- Classificação de relevância
  - Freqüência de termo
  - Freqüência de documento inversa
  - Relevância
  - Proximidade
- Recuperação baseada em semelhança
  - Modelo de espaço de vetor
  - Medida de semelhança do cosseno
  - Feedback por relevância
- Palavras de parada
- Relevância usando hiperlinks
  - Popularidade/prestígio
  - Transferência de prestígio
- PageRank
  - Modelo de caminhada aleatória
- Relevância baseada em texto de âncora
- Classificação por hub/autoridade
- Spamming do mecanismo de busca
- Sinônimos
- Homônimos
- Conceitos
- Consulta baseada em conceito
- Ontologias
- WordNet
- Índice invertido
- Falsa perda
- Falso negativo
- Falso positivo
- Precisão
- Rechamada
- Vasculhadores da Web
- Extração de informações
- Consulta de dados estruturados
- Resposta a perguntas
- Diretórios
- Hierarquia de classificação

Exercícios práticos

19.1 Calcule a relevância (usando definições apropriadas do termo freqüência e freqüência inversa do documento) de cada um dos Exercícios práticos neste capítulo com a consulta "relação SQL".

19.2 Suponha que você queira encontrar documentos que contenham pelo menos k de determinado conjunto de n palavras-chave. Suponha também que você tenha um índice de palavra-chave que lhe dê uma lista (classificada) de identificadores de documentos que contêm uma palavra-chave especificada. Dê um algoritmo eficiente para encontrar o conjunto de documentos desejado.

19.3 Proponha como implementar a técnica iterativa para calcular PageRank dado que a matriz T (mesmo na representação da lista de adjacência) não cabe na memória.

19.4 Proponha como um documento contendo uma palavra (como "leopardo") pode ser indexado de modo que seja recuperado com eficiência por consultas usando um conceito mais geral (como "carnívoro" ou "mamífero"). Você pode considerar que a hierarquia de conceitos não é muito profunda, de modo que cada conceito tenha apenas algumas generalizações (um conceito pode, no entanto, ter

uma grande quantidade de especializações). Você também pode considerar que receberá uma função que retorna o conceito para cada palavra em um documento.
Sugira também como uma consulta usando um conceito especializado pode recuperar documentos usando um conceito mais geral.

19.5 Suponha que listas invertidas sejam mantidas em blocos, com cada bloco anotando a maior classificação de popularidade e as notas TFIDF do documento nos blocos restantes da lista. Proponha como a mesclagem de listas invertidas pode parar mais cedo se o usuário quiser apenas as primeiras K respostas.

Exercícios

19.6 Usando uma definição simples do termo freqüência como o número de ocorrências do termo em um documento, dê as notas TFIDF de cada termo no conjunto de documentos consistindo neste e no próximo exercício.

19.7 Crie um pequeno exemplo com quatro pequenos documentos, cada um com um PageRank, e crie listas invertidas para os documentos, classificadas pelo PageRank. Você não precisa calcular o PageRank, mas apenas considerar alguns valores para cada página.

19.8 Suponha que você queira realizar uma consulta por palavra-chave em um conjunto de tuplas de um banco de dados, em que cada tupla tem apenas alguns poucos atributos, cada um contendo apenas algumas palavras. O conceito do termo freqüência faz sentido nesse contexto? E o de freqüência de documento inversa? Explique sua resposta. Sugira também como você pode definir a semelhança de duas tuplas usando os conceitos de TFIDF.

19.9 Os sites que querem obter alguma publicidade podem se juntar a um anel da Web, em que criam links para outros sites no anel, em troca de outros sites no anel criando links para o seu site. Qual é o efeito desses anéis sobre as técnicas de classificação de popularidade, como PageRank?

19.10 O mecanismo de busca Google oferece um recurso pelo qual os sites podem exibir anúncios fornecidos pelo Google. Os anúncios fornecidos são baseados no conteúdo da página. Sugira como o Google poderia escolher quais anúncios fornecer para uma página, dado o conteúdo da página.

19.11 Uma forma de criar uma versão específica para palavra-chave do PageRank é modificar o salto aleatório de modo que um salto só seja possível para páginas contendo a palavra-chave. Assim, as páginas que não contêm a palavra-chave, mas que estejam próximas (em termos de links) às páginas que contêm a palavra-chave também recebem uma classificação diferente de zero para essa palavra-chave.
   a. Dê equações definindo tal versão de PageRank específica para palavra-chave.
   b. Dê uma fórmula para calcular a relevância de uma página a uma consulta contendo várias palavras-chave.

19.12 A idéia de classificação de popularidade usando hiperlinks pode ser estendida para dados relacionais e XML, usando chave estrangeira e arestas IDREF no lugar de hiperlinks. Proponha como um esquema de classificação pode ter valor nas aplicações a seguir.
   a. Um banco de dados bibliográfico, que possui links de artigos para autores dos artigos e links para cada artigo que ele referencia.
   b. Um banco de dados de vendas que possui links de cada registro de vendas para os itens que foram vendidos.

   Sugira também por que a classificação de prestígio pode dar resultados menos significativos em um banco de dados de filmes, que registra qual ator atuou em quais filmes.

19.13 Qual é a diferença entre um falso positivo e uma falsa perda? Se for essencial que nenhuma informação relevante seja perdida por uma consulta de recuperação de informações, é aceitável ter falsos positivos ou falsas perdas? Por quê?

Notas bibliográficas

Chakrabarti [2002], Grossman e Frieder [20041, Witten *et al.* [1999] e Baeza-Yates e Ribeiro-Neto [1999] oferecem descrições em forma de livro-texto sobre a recuperação de informações. Chakrabarti [2002] oferece uma explicação detalhada dos vasculhadores da Web, técnicas de classificação e agrupamento e outras técnicas de exploração relacionadas à recuperação de informações. A indexação de documentos é explicada em detalhes por Witten *et al.* [1999]. Jones e Willet [1997] oferecem uma coleção de artigos sobre recuperação de informações. Salton [1989] é um antigo livro-texto sobre sistemas de recuperação de informações.

Brin e Page [1998] descrevem a anatomia do mecanismo de busca do Google, incluindo a técnica PageRank, enquanto uma técnica de classificação baseada em hubs e autoridades, chamada HITS, é descrita por Kleinberg [1999]. Bharat e Henzinger [1998] apresentam uma melhoria da técnica de classificação HITS. Essas e outras técnicas de classificação baseadas em popularidade (e técnicas para evitar o spamming do mecanismo de busca) são descritas com detalhes em Chakrabarti [2002]. Chakrabarti *et al.*

[1999] explica o vasculhamento focalizado da Web para encontrar páginas relacionadas a um assunto específico. Chakrabarti [1999] oferece um estudo sobre a descoberta de recursos Web.

O sistema Citeseer (`citeseer.ist.psu.edu`) mantém um banco de dados muito grande de publicações (artigos), com links de citação entre as publicações, e utiliza citações para classificar as publicações. Ele inclui uma técnica para ajustar a classificação da citação com base na idade de uma publicação, para compensar o fato de que as citações de uma publicação aumentam com o passar do tempo; sem o ajuste, documentos mais antigos costumam receber uma classificação mais alta do que verdadeiramente merecem.

A extração de informações e a resposta a perguntas têm tido um histórico bastante longo na comunidade de inteligência artificial. Jackson e Moulinier [2002] oferecem uma explicação em forma de livro-texto sobre a técnica de processamento da linguagem natural, enfatizando a extração de informações. Soderland [1999] descreve a extração de informações usando o sistema WHISK, enquanto Appelt e Israel [1999] oferecem um tutorial sobre extração de informações.

A Text Retrieval Conference (TREC) anual possui diversas trajetórias incluindo recuperação de documentos, resposta a perguntas, busca de genonomia e assim por diante. Cada trajetória define um problema e a infraestrutura para testar a qualidade das soluções para o problema. Os detalhes sobre a TREC podem ser encontrados em `trec.nist.gov`. As informações sobre a trajetória de resposta a perguntas podem ser encontradas em `trec.nist. gov/data/qa.html`.

Outras informações sobre WordNet podem ser encontradas em `wordnet.princeton.edu` e `glob-alwordnet.org`. O objetivo do sistema Cyc foi uma representação formal de grandes quantidades de conhecimento humano. Sua base de conhecimento contém uma grande quantidade de termos e as declarações sobre cada um. O Cyc também inclui um suporte para conhecimento e retirada de ambigüidade da linguagem natural. As informações sobre o sistema Cyc podem ser encontradas em cyc.com e opencyc.org.

Agrawal *et al.* [2002], Bhalotia *et al.* [2002] e Hristidis e Papakonstantinou [2002] abordam a consulta por palavra-chave dos dados relacionais. A consulta por palavra-chave dos dados XML é explicada por Florescu *et al.* [2000a] e Guo *et al.* [2003], entre outros.

Ferramentas

Google (`www.google.com`) atualmente é o mecanismo de busca mais popular, mas existem diversos outros mecanismos de busca, como MSN Search (`search.msn.com`) e Yahoo (`search.yahoo.com`). O site `searchenginewatch.com` oferece diversas informações sobre mecanismos de busca. Yahoo (`www.yahoo.com`) e o Open Directory Project (`dmoz.org`) oferecem hierarquias de classificação para sites.

PARTE 7

Arquitetura do sistema

A arquitetura de um sistema de banco de dados é bastante influenciada pelo sistema de computador básico em que o sistema de banco de dados é executado. Os sistemas de banco de dados podem ser centralizados, ou cliente-servidor, em que uma máquina servidora executa o trabalho em favor de várias máquinas clientes. Os sistemas de banco de dados também podem ser projetados para explorar arquiteturas paralelas de computador. Os bancos de dados distribuídos se espalham por várias máquinas separadas geograficamente.

O Capítulo 20 primeiro esboça as arquiteturas dos sistemas de banco de dados executando em sistemas servidores, que são usados em arquiteturas centralizadas e cliente-servidor. Os diversos processos que juntos implementam a funcionalidade de um banco de dados são esboçados aqui. O capítulo, então, esboça as arquiteturas de computador paralelas e as arquiteturas de banco de dados paralelas projetadas para diferentes tipos de computadores paralelos. Finalmente, o capítulo aborda as questões arquitetônicas na criação de um sistema de banco de dados distribuído.

O Capítulo 21 descreve como as diversas ações de um banco de dados, em particular o processamento da consulta, podem ser implementadas para explorar o processamento paralelo.

O Capítulo 22 apresenta uma série de questões que surgem em um banco de dados distribuído, e descreve como lidar com cada problema. Os problemas incluem como armazenar dados, como garantir a atomicidade das transações que executam em múltiplos sites, como realizar o controle de concorrência e como oferecer alta disponibilidade na presença de falhas. O processamento distribuído da consulta e os sistemas de diretório também são descritos nesse capítulo.

CAPÍTULO 20

Arquiteturas de sistema de banco de dados

A arquitetura de um sistema de banco de dados é bastante influenciada pelo sistema de computador básico em que ela trabalha, em particular, por aspectos da arquitetura de computador como redes, paralelismo e distribuição:

- As redes de computadores permitem que algumas tarefas sejam executadas em um sistema servidor e outras sejam executadas em sistemas cliente. Essa divisão de trabalho tem levado a *sistemas de banco de dados cliente-servidor*.
- O processamento paralelo dentro de um sistema de computador permite que as atividades do sistema de banco de dados sejam agilizadas, permitindo resposta mais rápida às transações, além de mais transações por segundo. As consultas podem ser processadas de um modo que explore o paralelismo oferecido pelo sistema de computador básico. A necessidade de processamento de consulta em paralelo levou a *sistemas de banco de dados paralelos*.
- A distribuição de dados pelos sites em uma organização permite que esses dados residam onde são gerados ou onde são mais necessários, mas ainda precisem ser acessados a partir de outros sites e de outros departamentos. Manter várias cópias do banco de dados em diferentes locais também permite que grandes organizações continuem suas operações de banco de dados mesmo quando um site é afetado por um desastre natural, como inundação, incêndio ou terremoto. Os *sistemas de banco de dados distribuídos* tratam de dados distribuídos geográfica e administrativamente, espalhados por diversos sistemas de banco de dados. Estudamos a arquitetura dos sistemas de banco de dados neste capítulo, começando com os sistemas centralizados tradicionais e abordando sistemas de banco de dados cliente-servidor, paralelos e distribuídos.

Arquiteturas centralizadas e cliente-servidor

Os sistemas de banco de dados centralizados são aqueles que executam em um único sistema de computador e não interagem com outros sistemas de computador. Esses sistemas de banco de dados se espalham por uma faixa desde sistemas de banco de dados monousuários executando em computadores pessoais até sistemas de banco de dados de alto desempenho, executando em sistemas servidores de alto nível. Os sistemas cliente-servidor, por outro lado, possuem a uncionalidade dividida entre um sistema servidor e vários sistemas cliente.

Sistemas centralizados

Um sistema de computador moderno, de uso geral, consiste em uma ou poucas CPUs e uma série de controladores de dispositivo que estão conectados por meio de um barramento comum, que oferece acesso à memória compartilhada (figura 20.1). As CPUs possuem memórias de cache local que armazenam cópias locais de partes da memória, para agilizar o acesso aos dados. Cada controlador de dispositivo está encarregado de um tipo de dispositivo específico (por exemplo, uma unidade de disco, um dispositivo de áudio ou uma tela de vídeo). As CPUs e os controladores de dispositivos podem executar simultaneamente, competindo pelo acesso à memória. A memória cache reduz a disputa pelo acesso à memória, pois reduz o número de vezes que a CPU precisa acessar a memória compartilhada.

Distinguimos duas maneiras como os computadores são usados: como sistemas monousuário e como sistemas multiusuário. Os computadores pessoais e as estações de trabalho caem na primeira categoria. Um **sistema monousuário**

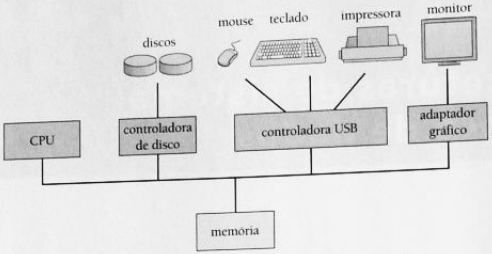


Figura 20.1 Um sistema de computador centralizado.

típico é uma unidade de desktop usada por uma única pessoa, normalmente com apenas uma CPU e um ou dois discos rígidos, e normalmente somente uma pessoa usando a máquina de cada vez. Um **sistema multiusuário** típico, por outro lado, tem mais discos e mais memória, pode ter várias CPUs e possui um sistema operacional multiusuário. Ele atende a uma grande quantidade de usuários que estão conectados ao sistema por meio de terminais.

Os sistemas de banco de dados para uso por usuários isolados normalmente não oferecem muitas das facilidades que um banco de dados multiusuário oferece. Em particular, eles podem não admitir o controle de concorrência, que não é exigido quando somente um único usuário pode gerar atualizações. Provisões para recuperação de falhas nesses sistemas são ausentes ou primitivas – por exemplo, elas podem consistir em simplesmente fazer um backup do banco de dados antes de qualquer atualização. Muitos desses sistemas não admitem SQL e oferecem uma linguagem de consulta mais simples, como uma variante da QBE. Ao contrário, os sistemas de banco de dados projetados para sistemas multiusuário admitem os recursos transacionais completos que estudamos anteriormente.

Embora os sistemas de computador de uso geral de hoje tenham vários processadores, eles possuem **paralelismo de granularidade grossa**, com somente alguns processadores (normalmente, cerca de dois a quatro), todos compartilhando a memória principal. Os bancos de dados sendo executados nessas máquinas normalmente não tentam particionar uma única consulta entre os processadores; em vez disso, eles executam cada consulta em um único processador, permitindo que várias consultas sejam executadas simultaneamente. Assim, esses sistemas admitem um throughput mais alto; ou seja, eles permitem um maior número de transações executando por segundo, embora as transações individuais não sejam executadas mais rapidamente.

Os bancos de dados projetados para máquinas de único processador já oferecem multitarefa, permitindo que vários processos sejam executados no mesmo processador em um modo de tempo compartilhado, dando uma visão ao usuário de vários processos rodando em paralelo. Assim, as máquinas paralelas com granularidade grossa parecem ser logicamente idênticas às máquinas de único processador, e os sistemas de banco de dados projetados para máquinas de tempo compartilhado podem ser facilmente adaptados para serem executados nelas.

Ao contrário, as máquinas com **paralelismo de granularidade fina** possuem uma grande quantidade de processadores, e os sistemas de banco de dados em execução em tais máquinas tentam colocar em paralelo tarefas isoladas (consultas, por exemplo) submetidas pelos usuários. Estudamos a arquitetura dos sistemas de banco de dados paralelos na seção "Sistemas paralelos".

Sistemas cliente-servidor

À medida que os computadores pessoais se tornaram mais rápidos, mais poderosos e mais baratos, houve um afastamento da arquitetura de sistema centralizada. Os computadores pessoais suplantaram os terminais conectados aos sistemas centralizados. De modo correspondente, os computadores pessoais assumiram a funcionalidade da interface com o usuário, que antes era tratada diretamente pelos sistemas centralizados. Como resultado, os sistemas centralizados de hoje atuam como **sistemas servidores**, que satisazem as solicitações geradas pelos *sistemas clientes*. A Figura 20.2 mostra a estrutura geral de um sistema cliente-servidor.

A funcionalidade oferecida pelos sistemas de banco de dados pode ser dividida de forma geral em duas partes – o front-end e o back-end. O back-end controla as estruturas de acesso, avaliação e otimização de consulta, controle de

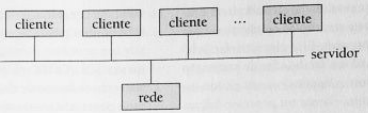


Figura 20.2 Estrutura geral de um sistema cliente-servidor.

concorrência e recuperação. O front-end de um sistema de banco de dados consiste nas ferramentas como a interface com o usuário da SQL, interfaces de formulário, ferramentas de geração de relatório e ferramentas de mineração e análise de dados. A interface entre o front-end e o back-end é por meio da SQL, ou por um programa de aplicação.

Padrões do tipo *ODBC* e *SQL*, que vimos no Capítulo 3, foram desenvolvidos para realizar a interface de clientes com servidores. Qualquer cliente que use a interface ODBC ou SQL pode se conectar a qualquer servidor que ofereça a interface.

Certos programas de aplicação, como planilhas e pacotes de análise estatística, utilizam a interface cliente-servidor diretamente para acessar dados de um servidor de back-end. Com efeito, eles oferecem front-ends especializados para tarefas em particular.

Os sistemas que lidam com grandes quantidades de usuários adotam uma arquitetura de três camadas, que vimos anteriormente na Figura 1.7 (Capítulo 1), em que o front-end é um navegador Web que se comunica com um servidor de aplicação. O servidor de aplicação, com efeito, atua como um cliente para o servidor de banco de dados.

Alguns sistemas de processamento de transação oferecem uma interface de **chamada de procedimento remoto transacional** para conectar clientes a um servidor. Essas chamadas parecem ser chamadas de procedimento comuns ao programador, mas todas as chamadas de procedimento remoto de um cliente são delimitadas em uma única transação na extremidade do servidor. Assim, se a transação for abortada, o servidor poderá desfazer os efeitos das chamadas de procedimento remoto individuais.

Arquiteturas de sistema servidor

Os sistemas servidores podem ser categorizados de modo geral como servidores de transação e servidores de dados.

- **Sistemas servidores de transação**, também denominados sistemas **servidores de consulta**, oferecem uma interface à qual os clientes podem enviar solicitações para realizar uma ação, em resposta às quais eles executam a ação e enviam os resultados de volta para o cliente. Normalmente, as máquinas clientes entregam transações aos sistemas servidores, onde essas transações são executadas, e os resultados são enviados de volta aos clientes que estão encarregados de exibir os dados. As solicitações podem ser especificadas pelo uso da SQL, ou por uma interface de programa de aplicação especializada.
- **Sistemas servidores de dados** permitem que os clientes interajam com os servidores, fazendo solicitações para ler ou atualizar dados, em unidades como arquivos ou páginas. Por exemplo, os servidores de arquivo oferecem uma interface de sistema de arquivo em que os clientes podem criar, atualizar, ler e excluir arquivos. Os servidores de dados para sistemas de banco de dados

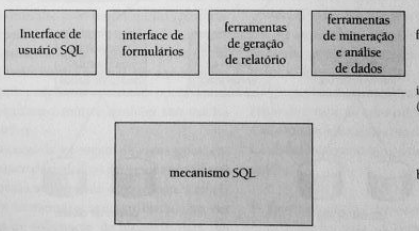


Figura 20.3 Funcionalidade de front-end e back-end.

oferecem muito mais funcionalidade: eles admitem unidades de dados – como páginas, tuplas ou objetos – que são menores do que um arquivo. Eles oferecem facilidades de indexação para dados e facilidades de transação de modo que os dados nunca fiquem em um estado inconsistente se uma máquina cliente ou processo falhar. Dessas, a arquitetura de servidor de transação por certo é a arquitetura mais utilizada. Vamos detalhar as arquiteturas de servidor de transação e servidor de dados na próxima e na seção "Servidores de dados".

Estrutura do processo servidor de transações

Um sistema típico de servidor de transação hoje consiste em vários processos acessando dados na memória compartilhada, como na Figura 20.4. Os processos que formam parte do sistema de banco de dados incluem

- Processos servidores: recebem consultas do usuário (transações), executam-nas e enviam os resultados de volta. As consultas podem ser submetidas aos processos servidores a partir de uma interface com o usuário, ou por um processo do usuário rodando a SQL embutida, ou via SQL, ODBC ou protocolos semelhantes. Alguns sistemas de banco de dados utilizam um processo separado para cada sessão do usuário, e alguns utilizam um único processo de banco de dados para todas as sessões do usuário, mas com vários threads, de modo que várias consultas possam ser executadas simultaneamente. (Um **thread** é como um processo, mas vários threads são executados como parte do mesmo processo, e todos os threads dentro de um processo são executados no mesmo espaço de memória virtual. Vários threads dentro de um processo podem ser executados simultaneamente.) Muitos sistemas de banco de dados utilizam uma arquitetura híbrida com vários processos, cada um executando vários threads.
- **Processo gerenciador de bloqueio:** implementa a funcionalidade do gerenciador de bloqueio, que inclui concessão de bloqueio, liberação de bloqueio e detecção de impasse.

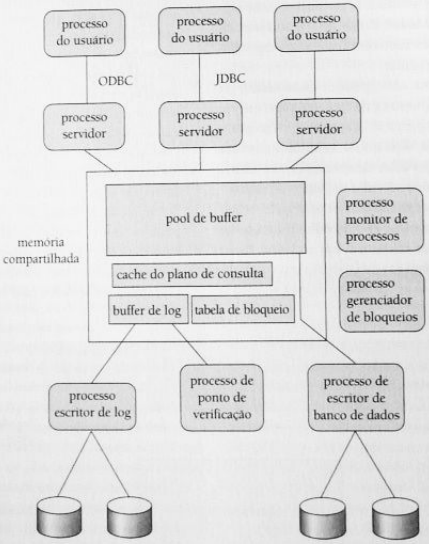


Figura 20.4 Memória compartilhada e estrutura de processos.

- **Processo escritor de banco de dados:** existem um ou mais processos que geram bloqueios de buffer modificados em disco de forma contínua.
- **Processo escritor de log:** gera registros de log a partir do buffer de registro de log para o armazenamento estável. Os processos servidores simplesmente acrescentam registros de log ao buffer de registro de logo na memória compartilhada, e se um log forçado for necessário, eles solicitam que o escritor de log processe para os registros de log de saída.
- **Processo de ponto de verificação:** esse processo realiza pontos de verificação periódicos.
- **Processo monitor de processos:** esse processo monitora outros processos, e se qualquer um deles falhar, são necessárias ações de recuperação para o processo, como abortar qualquer transação sendo executada pelo processo que falhou, e depois o reinício do processo.

A memória compartilhada contém todos os dados compartilhados, como:

- Pool de buffers
- Tabela de bloqueio
- Buffer de log, contendo registros de log aguardando para serem enviados ao log no armazenamento estável
- Planos de consulta em cache, que podem ser reutilizados se a mesma consulta for submetida novamente

Todos os processos do banco de dados podem acessar os dados na memória compartilhada. Como os vários processos podem ler ou realizar atualizações em estruturas de dados na memória compartilhada, é preciso haver um mecanismo para garantir que somente um deles esteja modificando qualquer estrutura de dados de uma só vez, e nenhum processo esteja lendo uma estrutura de dados enquanto ela está sendo escrita por outros. Essa **exclusão mútua** pode ser implementada por meio de funções do sistema operacional chamadas semáforos. Implementações alternativas, com menos sobrecarga, utilizam **instruções atômicas** especiais, admitidas pelo hardware do computador; um tipo de instrução atômica testa um local da memória e o define como 1 atomicamente. Outros detalhes de implementação da exclusão mútua podem ser encontrados em qualquer livro-texto padrão sobre sistema operacional. Os mecanismos de exclusão mútua também são usados para implementar latches.

Para evitar a sobrecarga da passagem de mensagens, em muitos sistemas de banco de dados, os processos servidores implementam o bloqueio atualizando diretamente a tabela de bloqueio (que está na memória compartilhada), em vez de enviar mensagens de solicitação de bloqueio para um processo gerenciador de bloqueio. O procedimento de solicitação de bloqueio executa as ações que o processo gerenciador de bloqueio tomaria ao apanhar uma solicitação de bloqueio. As ações sobre a solicitação e liberação de bloqueio são como aquelas na seção "Implementação do bloqueio" do Capítulo 16, mas com duas diferenças significativas:

- Como múltiplos processos de servidor podem acessar a memória compartilhada, a exclusão mútua precisa ser garantida sobre a tabela de bloqueio.
- Se um bloqueio não puder ser obtido imediatamente por causa de um conflito de bloqueio, o código de solicitação de bloqueio continua monitorando a tabela de bloqueio para verificar quando o bloqueio foi concedido. O código de liberação de bloqueio atualiza a tabela de bloqueio para observar qual processo recebeu o bloqueio.

Para evitar verificações repetidas sobre a tabela de bloqueio, semáforos do sistema operacional podem ser usados pelo código de solicitação de bloqueio para esperar uma notificação de concessão de bloqueio. O código de liberação de bloqueio precisa, então, usar o mecanismo de semáforo para notificar as transações aguardando de que seus bloqueios foram concedidos.

Mesmo que o sistema trate as solicitações de bloqueio pela memória compartilhada, ele ainda usa o processo gerenciador de bloqueio para a detecção de impasse.

Servidores de dados

Os sistemas servidores de dados são usados em redes locais, em que existe uma conexão de alta velocidade entre os clientes e o servidor, as máquinas clientes são comparáveis em poder de processamento às máquinas servidoras, e as tarefas a serem executadas exigem muita computação. Nesse tipo de ambiente, faz sentido enviar dados às máquinas clientes, para realizar todo o processamento na máquina cliente (o que pode levar algum tempo), e depois enviar os dados de volta à máquina servidora. Observe que essa arquitetura exige funcionalidade de back-end total nos clientes. As arquiteturas de servidor de dados têm sido particularmente populares nos sistemas de banco de dados orientados a objeto.

Questões interessantes surgem em tal arquitetura, pois o custo de tempo da comunicação entre o cliente e o servidor é alto em comparação com o de uma referência de memória local (milissegundos, *versus* menos de 100 nanossegundos):

- **Envio de página *versus* envio de item.** A unidade de comunicação para os dados pode ser de granularidade grossa, como uma página, ou de granularidade fina,

como uma tupla (ou um objeto, no contexto dos sistemas de banco de dados orientados a objeto). Usamos o termo **item** para nos referir a tuplas e objetos.

Se a unidade de comunicação for um único item, o overhead da passagem de mensagem é alto em comparação com a quantidade de dados transmitidos. Em vez disso, quando um item é solicitado, faz sentido também enviar outros itens que provavelmente são usados no futuro próximo. A busca de itens mesmo antes de serem solicitados é chamada de **busca prévia**. O envio de página pode ser considerado uma forma de busca prévia se vários itens residirem em uma página, pois todos os itens na página são enviados quando um processo deseja acessar um único item na página.

- **Bloqueio.** Os bloqueios normalmente são concedidos pelo servidor para itens de dados que ele envia às máquinas clientes. Uma desvantagem do envio de página é que as máquinas clientes podem receber bloqueios com granularidade muito grossa – um bloqueio sobre uma página implicitamente bloqueia todos os itens contidos na página. Mesmo que o cliente não esteja acessando alguns itens na página, ele implicitamente adquiriu bloqueios sobre todos os itens previamente buscados. Outras máquinas clientes que exigem bloqueios sobre esses itens podem ser bloqueadas desnecessariamente. Têm sido propostas técnicas para **desescalada** de bloqueio, em que o servidor pode solicitar a seus clientes que transfiram de volta os bloqueios sobre itens previamente buscados. Se uma máquina cliente não precisar de um item previamente buscado, ela pode transferir bloqueios sobre o item de volta ao servidor, e os bloqueios podem então ser alocados a outros clientes.
- **Caching de dados.** Os dados que são enviados a um cliente em favor de uma transação podem ser **colocados em cache** no cliente, mesmo depois que a transação terminar, se um espaço de armazenamento suficiente estiver disponível. Transações sucessivas no mesmo cliente podem ser capazes de utilizar os dados em cache. Porém, a **coerência de cache** é um problema: mesmo que uma transação encontre dados em cache, ela precisa ter certeza de que esses dados estão atualizados, pois eles podem ter sido atualizados por um cliente diferente depois que tiverem sido colocadas em cache. Assim, uma mensagem ainda precisa ser trocada com o servidor para verificar a validade dos dados e adquirir um bloqueio sobre eles.
- **Caching de bloqueio.** Se o uso de dados for, em sua maioria, particionado entre os clientes, ou seja, se eles solicitam dados que também são solicitados por outros clientes, os bloqueios também podem ser colocados em cache na máquina cliente. Suponha que um cliente encontre um item de dados no cache e que também encontre o bloqueio exigido para um acesso ao item de dados no cache. Então, o acesso pode prosseguir sem qualquer comunicação com o servidor. Porém, o servidor precisa registrar os bloqueios em cache; se um cliente solicitar um bloqueio do servidor, este precisará **chamar de volta** todos os bloqueios em conflito sobre o item de dados de quaisquer outras máquinas clientes que tenham colocado os bloqueios em cache. A tarefa se torna mais complicada quando as falhas de máquina são levadas em conta. Essa técnica difere da desescalada de bloqueio porque o caching de bloqueio ocorre entre as transações; caso contrário, as duas técnicas são semelhantes.

As referências bibliográficas oferecem mais informações sobre os sistemas de banco de dados cliente-servidor.

Sistemas paralelos

Sistemas paralelos melhoram as velocidades de processamento e E/S usando várias CPUs e discos em paralelo. As máquinas paralelas estão se tornando cada vez mais comuns, tornando o estudo de sistemas de banco de dados paralelos cada vez mais importante. A força motriz por trás dos sistemas de banco de dados paralelos é a demanda de aplicações que precisam consultar bancos de dados extremamente grandes (da ordem de terabytes – ou seja, 10^{12} bytes) ou que tenham de processar um número extremamente grande de transações por segundo (da ordem de milhares de transações por segundo). Os sistemas de banco de dados centralizados e cliente-servidor não são poderosos o suficiente para lidar com tais aplicações.

No processamento paralelo, muitas operações são realizadas simultaneamente, ao contrário do processamento serial, em que as etapas computacionais são realizadas sequencialmente. Uma máquina paralela com **granularidade grossa** consiste em um pequeno número de processadores poderosos; uma máquina **maciçamente paralela** ou **paralela com granularidade fina** utiliza milhares de processadores menores. A maior parte das máquinas de alto nível hoje em dia oferece algum grau de paralelismo com granularidade grossa: máquinas com dois ou quatro processadores são comuns. Computadores maciçamente paralelos podem ser distinguidos de máquinas paralelas com granularidade grossa pelo grau muito maior de paralelismo que eles admitem. Computadores paralelos, com centenas de CPUs e discos, estão disponíveis comercialmente.

Existem duas medidas principais de desempenho de um sistema de banco de dados: (1) **throughput**, o número de tarefas que podem ser completadas em determinado intervalo de tempo, e (2) **tempo de resposta**, a quantidade de tempo necessária para completar uma única tarefa desde o momento em que ela foi submetida. Um sistema que pro-

cessa um grande número de transações pequenas pode melhorar o throughput processando muitas transações em paralelo. Um sistema que processa grandes transações pode melhorar o tempo de resposta e também o throughput realizando subtarefas de cada transação em paralelo.

Ganho de velocidade e escala

Duas questões importantes no estudo do paralelismo são ganho de velocidade e ganho de escala. A execução de determinada tarefa em menos tempo e maior grau de paralelismo é chamada de **ganho de velocidade**. O tratamento de tarefas maiores pelo aumento do grau de paralelismo é denominado **ganho de escala**.

Considere uma aplicação de banco de dados rodando em um sistema paralelo com um certo número de processadores e discos. Agora, suponha que aumentemos o tamanho do sistema aumentando o número de processadores, discos e outros componentes do sistema. O objetivo é processar a tarefa no tempo inversamente proporcional ao número de processadores e discos alocados. Suponha que o tempo de execução de uma tarefa na máquina maior seja T_L, e que o tempo de execução da mesma tarefa na máquina menor seja T_S. O ganho de velocidade devido ao paralelismo é definido como T_S/T_L. Diz-se que o sistema paralelo demonstra **ganho de velocidade linear** se o ganho de velocidade for N quando o sistema maior tiver N vezes os recursos (CPU, disco etc.) do sistema menor. Se o ganho de velocidade for menor que N, diz-se que o sistema demonstra **ganho de velocidade sublinear**. A Figura 20.5 ilustra o ganho de velocidade linear e sublinear.

O ganho de escala se relaciona à capacidade de processar tarefas maiores na mesma quantidade de tempo, oferecendo mais recursos. Imagine que Q seja uma tarefa, e considere que Q_N seja uma tarefa que seja N vezes maior do que Q. Suponha que o tempo de execução da tarefa Q em determinada máquina M_S seja T_S, e que o tempo de execução da tarefa Q_N em uma máquina paralela M_L, que é N vezes maior que M_S, seja T_L. O ganho de escala é então definido como T_S/T_L. Diz-se que o sistema paralelo M_L demonstra **ganho de escala linear** na tarefa Q se $T_L = T_S$. Se $T_L > T_S$, diz-se que o sistema demonstra **ganho de escala sublinear**. A Figura 20.6 ilustra os ganhos de escala lineares e sublineares (em que os recursos aumentam em proporção ao tamanho do problema). Existem dois tipos de ganho de escala que são relevantes nos sistemas de banco de dados paralelos, dependendo de como o tamanho da tarefa é medido:

- No **ganho de escala em batch**, o tamanho do banco de dados aumenta e as tarefas, cujo tempo de execução depende do tamanho do banco de dados, são grandes. Um exemplo de tal tarefa é uma varredura de uma relação cujo tamanho é proporcional ao tamanho do banco de dados. Assim, o tamanho do banco de dados é a medida do tamanho do problema. O ganho de escala em batch também se aplica a aplicações científicas, como a execução de uma consulta em uma resolução N vezes mais fina ou a realização de uma simulação N vezes mais longa.
- No **ganho de escala de transação**, a velocidade com que as transações são submetidas ao banco de dados aumenta e o tamanho do banco de dados aumenta proporcionalmente à velocidade da transação. Esse tipo de ganho de escala é o que é relevante nos sistemas de processamento de transação, em que as transações são pequenas atualizações – por exemplo, um depósito ou saque de uma conta – e as velocidades de transação crescem quando mais contas são criadas. Esse processamento de transação é especialmente bem adaptado à execução paralela, pois as transações podem ser executadas de forma simultânea e independente em processadores paralelos, e cada transação leva aproximadamente o mesmo tempo, mesmo que o banco de dados cresça.

O ganho de escala normalmente é a métrica mais importante para medir a eficiência dos sistemas de banco de da-

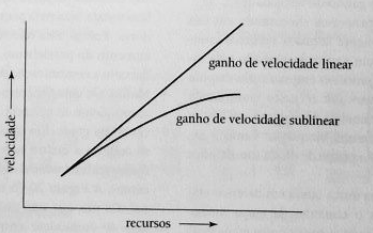


Figura 20.5 Ganho de velocidade com aumento de recursos.

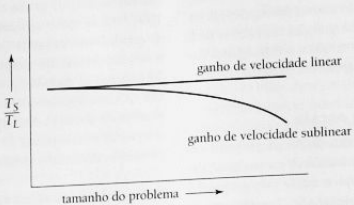


Figura 20.6 Ganho de velocidade com aumento do tamanho do problema e recursos.

dos paralelos. O objetivo do paralelismo nos sistemas de banco de dados normalmente é certificar-se de que o sistema de banco de dados possa continuar a trabalhar em uma velocidade aceitável, mesmo quando o tamanho do banco de dados e o número de transações aumenta. Aumentar a capacidade do sistema por meio do aumento do paralelismo oferece um caminho mais tranquilo para o crescimento de uma empresa do que a substituição de um sistema centralizado por uma máquina mais rápida (mesmo considerando que essa máquina exista). Porém, também temos de examinar os números de desempenho absolutos ao usar métricas de ganho de escala; uma máquina que ganha escala linearmente pode ter um desempenho pior do que uma máquina com ganho de escala sublinear, simplesmente porque esta última é muito mais rápida para dar partida.

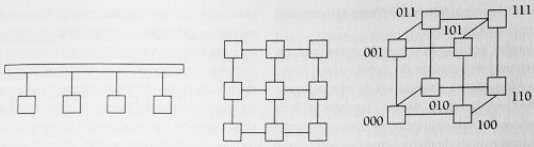
Diversos fatores atuam contra a operação paralela eficiente e podem diminuir o ganho de velocidade e o ganho de escala.

- **Custos de partida.** Existe um custo de partida associado ao início de um único processo. Em uma operação paralela consistindo em milhares de processos, o *tempo de partida* pode ofuscar o tempo de processamento real, afetando negativamente o ganho de velocidade.
- **Interferência.** Como os processos executando em um sistema paralelo normalmente acessam recursos compartilhados, pode haver um atraso ocasionado pela *interferência* de cada novo processo enquanto ele disputa com os processos existentes por recursos comumente mantidos, como um barramento do sistema, ou discos compartilhados, ou até mesmo bloqueios. Tanto o ganho de velocidade quanto o ganho de escala são afetados por esse fenômeno.
- **Distorção.** Dividindo uma única tarefa em diversas etapas paralelas, reduzimos o tamanho da etapa média. Apesar disso, o tempo de serviço para a única etapa mais lenta determinará o tempo de serviço para a tarefa como um todo. Normalmente, é difícil dividir uma tarefa em partes com tamanhos exatamente iguais, e o modo como os tamanhos são distribuídos, portanto, é *distorcido*. Por exemplo, se uma tarefa de tamanho 100 é dividia em 10 partes, e a divisão for distorcida, pode haver algumas tarefas com tamanho menor que 10 e outras com tamanho maior que 10; se até mesmo uma tarefa tiver tamanho 20, o ganho de velocidade obtido pela execução das tarefas em paralelo é de apenas cinco, em vez de dez, como esperaríamos.

Redes de interconexão

Sistemas paralelos consistem em um conjunto de componentes (processadores, memória e discos) que podem se comunicar entre si por meio de uma **rede de interconexão**. A Figura 20.7 mostra três tipos muito usados de redes de interconexão:

- **Barramento.** Todos os componentes do sistema podem enviar dados e receber dados de um único barramento de comunicação. Esse tipo de interconexão aparece na Figura 20.7a. O barramento poderia ser uma interconexão Ethernet ou paralela. As arquiteturas de barramento funcionam bem para pequenas quantidades de processadores. Porém, elas não se expandem muito bem com o aumento do paralelismo, pois o barramento só pode lidar com a comunicação de um componente de cada vez.
- **Malha.** Os componentes são nós em uma grade, e cada componente se conecta a todos os componentes adjacentes na grade. Em uma malha bidimensional, cada nó se conecta a quatro nós adjacentes, enquanto em uma malha tridimensional cada nó se conecta a seis nós adjacentes. A Figura 20.7b mostra uma malha bidimensional. Os nós que não estão diretamente conectados podem se comunicar entre si roteando mensagens por meio de uma seqüência de nós intermediários que estão



(a) barramento (b) malha (c) hipercubo

Figura 20.7 Redes de interconexão.

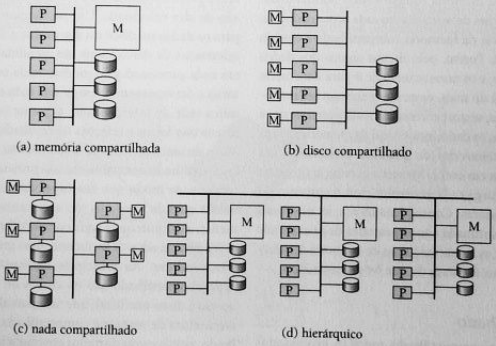
conectados diretamente um ao outro. O número de enlaces de comunicação cresce tanto quanto o número de componentes, e a capacidade de comunicação de uma malha, portanto, se expande melhor com o aumento do paralelismo.

- **Hipercubo.** Os componentes são numerados em binário, e um componente está conectado a outro se as representações binárias de seus números diferirem exatamente em um bit. Assim, cada um dos n componentes está conectado a $\log(n)$ outros componentes. A Figura 20.7c mostra um hipercubo com 8 nós. Em uma interconexão de hipercubo, uma mensagem de um componente pode alcançar qualquer outro componente passando por no máximo $\log(n)$ enlaces. Ao contrário, em uma arquitetura de malha, um componente pode estar a $2(\sqrt{n}-1)$ enlaces de distância de algum dos outros componentes (ou $\sqrt{n}$ enlaces de distância, se a interconexão de malha contornar as arestas da grade). Assim, os atrasos de comunicação em um hipercubo são muito menores o que em uma malha.

Arquiteturas paralelas de banco de dados

Existem vários modelos arquitetônicos para as máquinas paralelas. Entre os mais importantes estão aqueles da Figura 29.8 (na figura, M indica memória, P indica um processador e os discos aparecem como cilindros):

- **Memória compartilhada.** Todos os processadores compartilham uma memória comum (Figura 20.8a).
- **Disco compartilhado.** Todos os processadores compartilham um conjunto comum de discos (Figura 20.8b). Os sistemas de disco compartilhado às vezes são chamados de **clusters**.
- **Nada compartilhado.** Os processadores não compartilham nem uma memória comum nem um disco comum (Figura 20.8c).
- **Hierárquico.** Esse é um modelo híbrido das três arquiteturas anteriores (Figura 20.8d).



(a) memória compartilhada (b) disco compartilhado

(c) nada compartilhado (d) hierárquico

Figura 20.8 Arquiteturas de banco de dados paralelas.

Nas seções "Memória compartilhada" a "Hierárquico", elaboramos cada um desses modelos.

As técnicas usadas para agilizar o processamento da transação em sistemas de servidor de dados, como o caching de dados e bloqueio e a desescalada de bloqueio, esboçadas na seção "Servidores de dados", também podem ser usadas em bancos de dados paralelos de disco compartilhado, bem como em bancos de dados paralelos nada compartilhados. Na verdade, elas são muito importantes para o processamento eficiente da transação em tais sistemas.

Memória compartilhada

Em uma arquitetura de **memória compartilhada**, os processadores e discos possuem acesso a uma memória comum, normalmente por meio de um barramento ou por meio de uma rede de interconexão. A memória compartilhada é uma forma de comunicação extremamente eficaz entre os processadores – os dados na memória compartilhada podem ser acessados por qualquer processador sem serem movidos com o software. Um processador pode enviar mensagens a outros processadores muito mais rapidamente usando escritas na memória (que normalmente levam menos de um microssegundo) do que enviando uma mensagem por um mecanismo de comunicação. A desvantagem das máquinas de memória compartilhada é que a arquitetura não é expansível além de 32 ou 64 processadores, já que o barramento ou a rede de interconexão se torna um gargalo (por ser compartilhado por todos os processadores). A inclusão de mais processadores não ajuda depois de um ponto, pois os processadores gastarão a maior parte do seu tempo aguardando sua vez no barramento para acessar a memória.

As arquiteturas de memória compartilhada normalmente possuem grandes caches de memória em cada processador, de modo que a referência da memória compartilhada é evitada sempre que possível. Porém, pelo menos alguns dos dados não estarão no cache, e os acessos terão de ir para a memória compartilhada. Além do mais, os caches precisam ser mantidos coerentes; ou seja, se um processador realiza uma escrita a um local da memória, os dados nesse local da memória devem ser atualizados ou removidos de qualquer processador em que os dados estejam em cache. Manter a coerência do cache torna-se uma sobrecarga cada vez maior com o aumento do número de processadores. Conseqüentemente, as máquinas com memória compartilhada não são capazes de se expandir além de certo ponto; as atuais máquinas de memória compartilhada não podem aceitar mais do que 64 processadores.

Disco compartilhado

No modelo de **disco compartilhado**, todos os processadores podem acessar todos os discos diretamente por meio de uma rede de interconexão, mas os processadores possuem memórias privadas. Existem duas vantagens nessa arquitetura em relação à arquitetura de memória compartilhada. Primeiro, como cada processador tem sua própria memória, o barramento de memória não é um gargalo. Em segundo lugar, é um modo barato de oferecer um grau de **tolerância a falhas**. Se um processador (ou sua memória) falhar, os outros processadores podem assumir suas tarefas, pois o banco de dados é residente nos discos que são acessíveis por todos os processadores. Podemos tornar o próprio subsistema de disco tolerante a falhas usando uma arquitetura RAID, conforme descrevemos no Capítulo 11. A arquitetura de disco compartilhado encontrou aceitação em muitas aplicações.

O problema principal com um sistema de disco compartilhado novamente é o de expansão. Embora o barramento de memória não seja mais um gargalo, a interconexão com o subsistema de disco agora é; isso acontece particularmente em uma situação em que o banco de dados faz uma grande quantidade de acessos aos discos. Em comparação com sistemas de memória compartilhada, os sistemas de disco compartilhado podem se expandir até um número um tanto maior de processadores, mas a comunicação pelos processadores é mais lenta (até alguns milissegundos na ausência de hardware de uso especial para comunicação), pois precisa passar por uma rede de comunicação.

Nada compartilhado

Em um sistema **nada compartilhado**, cada nó da máquina consiste em um processador, memória e um ou mais discos. Os processadores em um nó podem se comunicar com outro processador em outro nó por uma rede de interconexão de alta velocidade. Um nó funciona como o servidor para os dados no disco ou discos que o nó possui. Como as referências de disco locais são atendidas por discos locais em cada processador, o modelo nada compartilhado contorna a desvantagem de exigir que toda a E/S passe por uma única rede de interconexão; somente consultas, acessos a discos não locais e relações de resultado passam pela rede. Além do mais, as redes de interconexão para sistemas nada compartilhado normalmente são projetadas para serem escaláveis, de modo que sua capacidade de transmissão aumenta quando mais nós são acrescentados. Conseqüentemente, as arquiteturas nada compartilhado são mais escaláveis e podem admitir facilmente uma grande quantidade de processadores. As principais desvantagens dos sistemas nada compartilhado são os custos de comunicação e do acesso a disco não local, que são mais altos do que em uma arquitetura de memória compartilhada ou disco compartilhado, pois o envio de dados envolve a interação do software nas duas extremidades.

Hierárquico

A **arquitetura hierárquica** combina as características das arquiteturas de memória compartilhada, disco compartilhado e nada compartilhado. No nível superior, o sistema consiste em nós que estão conectados por uma rede de interconexão e não compartilham discos ou memória uns com os outros. Assim, o nível superior é uma arquitetura do tipo nada compartilhado. Cada nó do sistema poderia realmente ser um sistema de memória compartilhada com alguns processadores. Como alternativa, cada nó poderia ser um sistema de disco compartilhado, e cada um dos sistemas compartilhando um conjunto de discos poderia ser um sistema de memória compartilhada. Assim, um sistema poderia ser montado como uma hierarquia, com arquitetura de memória compartilhada, com alguns processadores na base e uma arquitetura do tipo nada compartilhado no topo, com possivelmente uma arquitetura de disco compartilhado no meio. A Figura 20.8d ilustra uma arquitetura hierárquica com nós de memória compartilhada conectados em uma arquitetura do tipo nada compartilhado. Os sistemas de banco de dados paralelos de hoje executam em várias dessas arquiteturas.

As tentativas de reduzir a complexidade da programação de tais sistemas têm gerado arquiteturas de **memória virtual distribuída**, em que logicamente existe uma única memória compartilhada, mas fisicamente existem vários sistemas de memória disjuntos; o hardware de mapeamento de memória virtual, junto com o software do sistema, permite que cada processador veja as memórias disjuntas como uma única memória virtual. Como as velocidades de acesso são diferentes, dependendo se a página está disponível no local ou não, tal arquitetura também é conhecida como **arquitetura de memória não uniforme** (NUMA – Non Uniform Memory Architecture).

Sistemas distribuídos

Em um **sistema de banco de dados distribuído**, o banco de dados é armazenado em vários computadores. Os computadores em um sistema distribuído se comunicam entre si através de diversos meios de comunicação, como redes de alta velocidade ou linhas telefônicas. Eles não compartilham memória principal ou discos. Os computadores em um sistema distribuído podem variar em tamanho e função, desde estações de trabalho até sistemas de mainframe.

Os computadores em um sistema distribuído são referenciados por uma série de nomes diferentes, como **sites** ou **nós**, dependendo do contexto em que são mencionados. Usamos principalmente o termo **site** para enfatizar a distribuição física desses sistemas. A estrutura geral de um sistema distribuído aparece na Figura 20.9.

As principais diferenças entre os bancos de dados paralelos nada compartilhado e os bancos de dados distribuídos são que os bancos de dados distribuídos normalmente estão separados geograficamente, são administrados separadamente e possuem uma interconexão mais lenta. Outra diferença importante é que, em um sistema de banco de dados distribuído, diferenciamos entre transações locais e globais. Uma **transação local** é aquela que acessa dados apenas de sites em que a transação foi iniciada. Uma **transação global**, por outro lado, é aquela que acessa dados em um site diferente daquele em que a transação foi iniciada, ou acessa dados em vários sites diferentes.

Existem vários motivos para se criar sistemas de banco de dados distribuídos, incluindo o compartilhamento de dados, autonomia e disponibilidade.

- **Compartilhamento de dados.** A principal vantagem da criação de um sistema de banco de dados distribuído é a provisão de um ambiente em que os usuários em um site

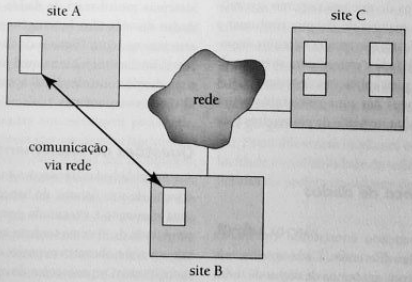


Figura 20.9 Um sistema distribuído.

podem ser capazes de acessar os dados residindo em outros sites. Por exemplo, em um sistema bancário distribuído, em que cada agência armazena dados relativos a si própria, é possível que um usuário em uma agência acesse dados em outra agência. Sem essa capacidade, um usuário que deseja transferir fundos de uma agência para outra teria de lançar mão de algum mecanismo externo que acoplaria sistemas existentes.

- **Autonomia.** A principal vantagem do compartilhamento de dados por meio da distribuição de dados é que cada site é capaz de reter um grau de controle sobre os dados que estão armazenados localmente. Em um sistema centralizado, o administrador de banco de dados do site central controla o banco de dados. Em um sistema distribuído, existe um administrador de banco de dados global, responsável pelo sistema inteiro. Uma parte dessas responsabilidades é delegada ao administrador de banco de dados local para cada site. Dependendo do projeto do sistema de banco de dados distribuído, cada administrador pode ter um grau de **autonomia local** diferente. A possibilidade de autonomia local normalmente é uma grande vantagem dos bancos de dados distribuídos.
- **Disponibilidade.** Se um site falhar em um sistema distribuído, os sites restantes podem ser capazes de continuar operando. Em particular, se os itens de dados forem **replicados** em vários sites, uma transação precisando de um item de dados em particular poderá encontrar esse item em um dentre vários sites. Assim, a falha de um site não implica necessariamente o desligamento do sistema.

  A falha de um site precisa ser detectada pelo sistema, e pode ser preciso tomar a ação apropriada para se recuperar da falha. O sistema não pode mais usar os serviços do site que falhou. Finalmente, quando o site que falhou se recuperar ou for reparado, é preciso haver mecanismos para integrá-lo tranquilamente de volta ao sistema.

  Embora a recuperação de falha seja mais complexa nos sistemas distribuídos do que nos sistemas centralizados, a capacidade da maioria do sistema continuar a operar apesar da falha de um site resulta na maior disponibilidade. A disponibilidade é crucial para sistemas de banco de dados usados para aplicações de tempo real. A perda de acesso aos dados por uma companhia aérea, por exemplo, pode resultar na perda de passageiros para os concorrentes.

Exemplo de um banco de dados distribuído

Considere um sistema bancário consistindo em quatro agências em quatro cidades diferentes. Cada agência tem seu próprio computador, com um banco de dados de todas as contas mantidas nessa agência. Cada instalação desse tipo, portanto, é um site. Há também um único site que mantém informações sobre todas as agências do banco. Cada agência mantém (entre outras) uma relação *conta*(*Esquema_conta*), em que

$$Esquema\_conta = (numero\_conta, nome\_agência, saldo)$$

O site contendo informações sobre todas as agências do banco mantém a relação *agência*(*Esquema_agência*), onde

$$Esquema\_agência = (nome\_agência, cidade\_agência, ativos)$$

Existem outras relações mantidas nos diversos sites; nós as ignoramos para os fins do nosso exemplo.

Para ilustrar a diferença entre os dois tipos de transações – locais e globais – nos sites, considere uma transação para somar $50 ao número de conta A-177 localizado na agência Valleyview. Se a transação foi iniciada na agência Valleyview, então ela é considerada local; caso contrário, ela é considerada global. Uma transação para transferir $50 da conta A-177 para a conta A-305, que está localizada na agência Hillside, é uma transação global, pois as contas nos dois sites diferentes são acessadas como resultado de sua execução.

Em um sistema de banco de dados distribuído ideal, os sites compartilhariam um esquema global comum (embora algumas relações possam ser armazenadas apenas em alguns sites), todos os sites executariam o mesmo software de gerenciamento de banco de dados distribuído, e os sites estariam mais cientes da existência um do outro. Se um banco de dados distribuído for construído do zero, realmente seria possível conseguir os tais objetivos. Porém, na realidade, um banco de dados distribuído precisa ser construído pelo vínculo de vários sistemas de banco de dados já existentes, cada um com seu próprio esquema e possivelmente executando diferentes softwares de gerenciamento de banco de dados. Esses sistemas às vezes são chamados **sistemas multibanco de dados** ou **sistemas de banco de dados distribuídos heterogêneos**. Discutimos sobre esses sistemas na seção "Bancos de dados distribuídos heterogêneos" do Capítulo 22, na qual mostramos como conseguir um grau de controle global apesar da heterogeneidade dos sistemas componentes.

Questões de implementação

A atomicidade das transações é uma questão importante na criação de um sistema de banco de dados distribuído. Se uma transação é executada por dois sites, a menos que os projetistas do sistema tenham cuidado, ela pode confirmar em um site e abortar em outro, ocasionando um estado inconsistente. Os protocolos de commit de transação garantem que tal situação não pode surgir. O *protocolo de commit*

em duas fases (2PC – 2 Phase Commit) é o mais utilizado desses protocolos.

A idéia básica por trás do 2PC é que cada site execute a transação até imediatamente antes do commit, e depois deixe a decisão de commit para um único site coordenador; diz-se que a transação está no estado *pronto* em um site nesse ponto. O coordenador decide confirmar a transação somente se ela alcançar o estado pronto em cada site em que é executada; caso contrário (por exemplo, se a transação abortar em qualquer site), o coordenador decide abortar a transação. Cada site em que a transação executou precisa seguir a decisão do coordenador. Se um site falhar quando uma transação estiver no estado pronto, quando o site se recuperar da falha, ele deverá estar em posição para confirmar ou abortar a transação, dependendo da decisão do coordenador. O protocolo 2PC é descrito com detalhes na seção "Commit de duas fases" do Capítulo 22.

O controle de concorrência é outro problema em um banco de dados distribuído. Como uma transação pode acessar itens de dados em vários sites, os gerenciadores de transação em vários sites podem ter de se coordenar para implementar o controle de concorrência. Se o bloqueio for usado (como quase sempre acontece na prática), ele pode ser realizado localmente nos sites contendo os itens de dados acessados, mas há também uma possibilidade de impasse envolvendo transações com origem em vários sites. Portanto, a detecção de impasse precisa ser executada por vários sites. As falhas são mais comuns nos sistemas distribuídos, pois não apenas os computadores podem falhar, mas os enlaces de comunicação também. A replicação de itens de dados, que é a chave para a funcionalidade contínua dos bancos de dados distribuídos quando existem falhas, complica ainda mais o controle de concorrência. A seção "Controle de concorrência em bancos de dados distribuídos" do Capítulo 22 oferece cobertura detalhada do controle de concorrência nos bancos de dados distribuídos.

Os modelos de transação padrão, baseados em várias ações executadas por uma única unidade de programa, normalmente são impróprios para a execução de tarefas que cruzam as fronteiras de bancos de dados que não podem ou não cooperarão para implementar protocolos como 2PC. Técnicas alternativas, baseadas em *mensagens persistentes* para a comunicação, geralmente são usadas para tais tarefas.

Quando as tarefas a serem executadas são complexas, envolvendo vários bancos de dados e/ou várias interações com humanos, a coordenação das tarefas e a garantia das propriedades da transação para as tarefas se tornam mais complicadas. *Sistemas de gerenciamento de fluxo de trabalho* são criados para ajudar na execução de tais tarefas. A seção "Modelos alternativos de processamento de transação" do Capítulo 22 descreve as mensagens persistentes, enquanto a seção "Fluxos de trabalho transacionais" do Capítulo 25 descreve os sistemas de gerenciamento de fluxo de trabalho.

Caso uma organização tenha de escolher entre uma arquitetura distribuída e uma arquitetura centralizada para implementar uma aplicação, o arquiteto do sistema precisa considerar as vantagens e as desvantagens da distribuição de dados. Já vimos as vantagens do uso de bancos de dados distribuídos. A principal desvantagem dos sistemas de banco de dados distribuídos é a complexidade adicional exigida para garantir a coordenação correta entre os sites. Essa maior complexidade tem diversas formas:

- **Custo de desenvolvimento de software.** É mais difícil implementar um sistema de banco de dados distribuído; assim, ele é mais dispendioso.
- **Maior potencial para bugs.** Como os sites que constituem o sistema distribuído operam em paralelo, é mais difícil garantir a exatidão dos algoritmos, especialmente a operação durante falhas de parte do sistema, e a recuperação de falhas. Existe o potencial para bugs extremamente sutis.
- **Maior sobrecarga de processamento.** A troca de mensagens e o cálculo adicional exigido para alcançar a coordenação entre os sites são uma forma de sobrecarga que não acontece nos sistemas centralizados. Existem várias técnicas para o projeto de banco de dados distribuído, variando desde projetos totalmente distribuídos até aqueles que incluem um alto grau de centralização. Vamos estudá-los no Capítulo 22.

Tipos de redes

Os bancos de dados distribuídos e os sistemas cliente-servidor são montados em torno de redes de comunicação. Existem basicamente dois tipos de redes: **redes locais e redes remotas**. A diferença principal entre as duas é o modo como são distribuídas geograicamente. Em redes locais, os processadores são distribuídos por pequenas áreas geográficas, como um único prédio ou uma série de prédios adjacentes. Em redes remotas, por outro lado, diversos processadores autônomos são distribuídos por uma grande área geográfica (como os Estados Unidos ou o mundo inteiro). Essas diferenças implicam em grandes variações na velocidade e confiabilidade da rede de comunicação, e são refletidas no projeto de sistema operacional distribuído.

Redes locais

As **redes locais** (LANs – Local Area Networks) (Figura 20.10) surgiram no início da década de 1970 como um meio para os computadores se comunicarem e comparti-

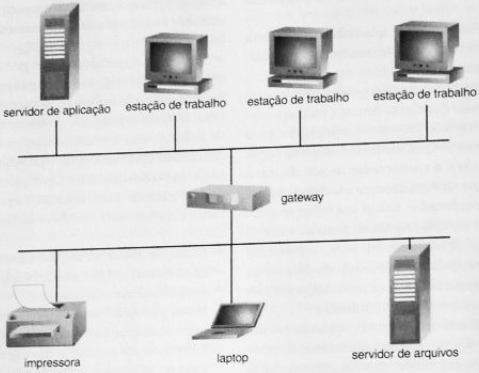


Figura 20.10 Rede local.

lharem dados entre si. As pessoas reconheceram que, para muitas empresas, diversos computadores pequenos, cada um com suas próprias aplicações autocontidas, são mais econômicos do que um único sistema grande. Como cada computador pequeno provavelmente precisará acessar um complemento total de dispositivos periféricos (como discos e impressoras), e como alguma forma de compartilhamento de dados provavelmente ocorrerá em uma única empresa, foi um passo natural conectar esses pequenos sistemas a uma rede.

As LANs geralmente são usadas em um ambiente de escritório. Todos os sites nesses sistemas estão próximos um do outro, de modo que os enlaces de comunicação costumam ter uma velocidade maior e uma taxa de erros menor que seus correspondentes nas redes remotas. Os enlaces mais comuns em uma rede local são par trançado, cabo coaxial, fibra óptica e, cada vez mais, conexões sem fio. As velocidades de comunicação variam desde alguns megabits por segundo (para redes locais sem fio), até 1 gigabit por segundo para Gigabit Ethernet. A Ethernet padrão trabalha a 10 megabits por segundo, enquanto a Fast Ethernet é capaz de transmitir 100 megabits por segundo.

Uma **rede de área de armazenamento** (SAN – Storage Area Network) é um tipo especial de rede local de alta velocidade projetada para conectar grandes bancos de dispositivos de armazenamento (discos) a computadores que utilizam os dados (ver Figura 20.11). Assim, as redes de área de armazenamento ajudam a construir *sistemas de disco compartilhado* em grande escala. A motivação para o uso de redes de área de armazenamento para conectar vários computadores a grandes bancos de dispositivos de armazenamento é basicamente a mesma que para os bancos de dados de disco compartilhado, a saber

- Escalabilidade pelo acréscimo de mais computadores
- Alta disponibilidade, pois os dados ainda são acessíveis mesmo que um computador falhe

Organizações RAID são utilizadas nos dispositivos de armazenamento para assegurar a alta disponibilidade dos dados, permitindo que o processamento continue mesmo que os discos individuais falhem. As redes de área de armazenamento normalmente são montadas com redundância, como vários caminhos entre os nós, de modo que, se um componente como um enlace ou uma conexão com a rede falhar, a rede continuará funcionando.

Redes remotas

As **redes remotas** (WANs – Wide Area Networks) surgiram no final da década de 1960, principalmente como um projeto de pesquisa acadêmica para oferecer comunicação eficiente entre os sites, permitindo que o hardware e o software sejam compartilhados de modo conveniente e econômico por uma grande comunidade de usuários. Os sistemas que permitiam que terminais remotos fossem conectados a um computador central por linhas telefônicas foram desenvolvidos no início dos anos 60, mas não eram verdadeiras WANs. A

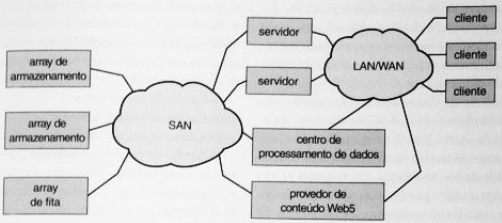


Figura 20.11 Rede de área de armazenamento.

primeira WAN a ser projetada e desenvolvida foi a *Arpanet*. O trabalho na Arpanet foi iniciado em 1968. A Arpanet cresceu a partir de uma rede experimental de quatro sites até uma rede mundial de redes, a **Internet**, compreendendo centenas de milhões de sistemas de computador. Os enlaces típicos na Internet são linhas de fibra óptica e, às vezes, canais de satélite. As taxas de dados para enlaces remotos normalmente variam de alguns megabits por segundo até centenas de gigabits por segundo. O último enlace, os sites de usuário final, normalmente é baseado na tecnologia de *linha de assinante digital* (DSL – Digital Subscriber Line) (aceitando alguns megabits por segundo) ou modem a cabo (aceitando 10 megabits por segundo), ou conexões de modem por linhas telefônicas (admitindo até 56 kilobits por segundo). As WANs podem ser classificadas em dois tipos:

- Nas WANs de **conexão descontínua**, como aquelas baseadas em conexões sem fio, os hosts estão conectados à rede apenas em parte do tempo.
- Nas WANs de **conexão contínua**, como a Internet a cabo, os hosts estão conectados à rede o tempo todo.

As redes que não estão conectadas continuamente em geral não permitem transações entre sites, mas podem manter cópias locais de dados remotos e atualizar as cópias periodicamente (a cada noite, por exemplo). Para aplicações em que a consistência não é crítica, como no compartilhamento de documentos, os sistemas de groupware, como Lotus Notes, permitem que as atualizações de dados remotos sejam feitas localmente, e as atualizações são propagadas de volta ao site remoto periodicamente. Existe um potencial para atualizações em conflito em diferentes sites, conflitos que precisam ser detectados e resolvidos. Um mecanismo para detectar atualizações em conflito é descrito mais adiante, na seção "Desconectividade e consistência" do Capítulo 24; o mecanismo de resolução para atualizações em conflito, porém, depende da aplicação.

Resumo

- Os sistemas de banco de dados centralizados são executados inteiramente em um único computador. Com o crescimento dos computadores pessoais e das redes locais, a funcionalidade do ront-end de banco de dados passou cada vez mais para os clientes, com os sistemas servidores oferecendo a funcionalidade do back-end. Os protocolos de interace cliente-servidor ajudaram no crescimento dos sistemas de banco de dados cliente-servidor.
- Os servidores podem ser de transação ou de dados, embora o uso de servidores de transação seja bem mais comum para oferecer serviços de banco de dados.
  - Os servidores de transação possuem vários processos, possivelmente rodando em vários processadores. Para que esses processos tenham acesso a dados comuns, como o buffer de banco de dados, os sistemas armazenam tais dados na memória compartilhada. Além dos processos que lidam com consultas, existem processos do sistema que executam tarefas como o gerenciamento de bloqueio e log e o uso de ponto de verificação.
  - Os sistemas servidores de dados oferecem dados brutos aos clientes. Esses sistemas lutam para minimizar a comunicação entre clientes e servidores, colocando dados e bloqueios em cache nos clientes. Os sistemas de banco de dados paralelos utilizam otimizações semelhantes.
- Os sistemas de banco de dados paralelos consistem em vários processadores e vários discos conectados por uma rede de interconexão rápida. O ganho de velocidade mede o quanto podemos aumentar a velocidade de processamento aumentando o paralelismo, para uma única transação. O aumento de escala mede como podemos tratar de um número maior de transações aumentando o paralelismo. Interferência, distorção e custos de partida atuam como barreiras para se conseguir ganho de velocidade e ganho de escala ideais.

- Algumas arquiteturas de banco de dados paralelas são memória compartilhada, disco compartilhado, nada compartilhado e arquiteturas hierárquicas. Essas arquiteturas possuem diferentes opções de escalabilidade *versus* velocidade de comunicação.
- Um sistema de banco de dados distribuído é uma coleção de sistemas de banco de dados parcialmente independentes que (de forma ideal) compartilham um esquema comum e coordenam o processamento de transações que acessam dados não locais. Os sistemas se comunicam um com o outro por meio de uma rede de comunicação que trata do roteamento e de estratégias de conexão.
- Principalmente, existem dois tipos de redes de comunicação: redes locais e redes remotas. As redes locais conectam nós que são distribuídos por pequenas áreas geográficas, como um único prédio ou alguns prédios adjacentes. As redes remotas conectam nós espalhados por uma grande área geográfica. A Internet é a rede remota mais utilizada hoje em dia.

  As redes de área de armazenamento são um tipo especial de rede local, projetado para oferecer a interconexão rápida entre grandes bancos de dispositivos de armazenamento e vários computadores.

Termos de revisão

- Sistemas centralizados
- Sistemas servidores
- Paralelismo de granularidade grossa
- Paralelismo de granularidade fina
- Estrutura de processo de banco de dados
- Exclusão mútua
- Thread
- Processos servidores
  - Processo gerenciador de bloqueio
  - Processo escritor de banco de dados
  - Processo escritor de log
  - Processo de ponto de verificação
  - Processo monitor de processos
- Sistemas cliente-servidor
- Servidor de transações
- Servidor de consulta
- Servidor de dados
  - Busca prévia
  - Desescalada
  - Caching de dados
  - Coerência de cache
  - Caching de bloqueio
  - Chamar de volta
- Sistemas paralelos
- Throughput
- Tempo de resposta
- Ganho de velocidade
  - Ganho de velocidade linear
  - Ganho de velocidade sublinear
- Ganho de escala
  - Ganho de escala linear
  - Ganho de escala sublinear
  - Ganho de escala em batch
  - Ganho de escala de transação
- Custos de partida
- Interferência
- Distorção
- Redes de interconexão
  - Barramento
  - Malha
  - Hipercubo
- Arquiteturas de banco de dados paralelas
  - Memória compartilhada
  - Disco compartilhado (clusters)
  - Nada compartilhado
  - Hierárquico
- Tolerância a falhas
- Memória virtual distribuída
- Arquitetura de memória não uniforme (NUMA)
- Sistemas distribuídos
- Banco de dados distribuído
  - Sites (nós)
  - Transação local
  - Transação global
  - Autonomia local
- Sistemas multibanco de dados
- Tipos de rede
  - Redes locais (LAN)
  - Redes remotas (WAN)
  - Redes de área de armazenamento (SAN)

Exercícios práticos

20.1 Em vez de armazenar estruturas compartilhadas na memória compartilhada, uma arquitetura alternativa seria armazená-las na memória local de um processo especial e acessar os dados compartilhados pela comunicação interprocessual com o processo. Qual seria a desvantagem de tal arquitetura?

20.2 Em sistemas cliente-servidor típicos, a máquina servidora é muito mais poderosa do que os clientes; ou seja, seu processador é mais rápido, ele pode ter vários processadores e tem mais memória e capacidade de disco. Considere, em vez disso, um cenário em que as máquinas cliente e servidora possuem exatamente a mesma potência. Faria sentido montar um sistema cliente-servidor nesse cenário? Por

quê? Que cenário seria mais adequado a uma arquitetura de servidores de dados?

20.3 Considere um sistema de banco de dados orientado a objeto baseado em uma arquitetura cliente-servidor, com o servidor atuando como um servidor de dados.

a. Qual é o efeito da velocidade da interconexão entre o cliente e o servidor sobre a escolha entre o envio de objeto e página?

b. Se o envio de página for utilizado, o cache de dados no cliente pode ser organizado como um cache de objeto ou como um cache de página. O cache de página armazena dados em unidades de uma página, enquanto o cache de objeto armazena dados em unidades de objetos. Considere que os objetos são menores do que uma página. Descreva um benefício de um cache de objeto em relação a um cache de página.

20.4 Suponha que uma transação seja escrita em C com SQL embutida, e cerca de 80% do tempo seja gasto no código SQL, com os 20% restantes gastos no código C. Quanto ganho de velocidade pode-se esperar alcançar se o paralelismo for usado apenas para o código SQL? Explique.

20.5 Considere uma rede baseada em linhas telefônicas dial-up, em que os sites se comunicam periodicamente, por exemplo, a cada noite. Essas redes normalmente são configuradas com um site servidor e vários sites clientes. Os sites clientes se conectam apenas ao servidor, e trocam dados com outros clientes armazenando dados no servidor e recuperando dados armazenados no servidor por outros clientes. Qual é a vantagem de tal arquitetura em relação a outra em que um site só pode trocar dados com outro site discando para ele primeiro?

Exercícios

20.6 Por que é relativamente fácil portar um banco de dados de uma máquina de único processador para uma máquina de multiprocessador se as consultas individuais não precisam ser executadas em paralelo?

20.7 As arquiteturas de servidor de transações são populares para bancos de dados relacionais cliente-servidor, em que as transações são curtas. Por outro lado, as arquiteturas de servidor de dados são populares para sistemas de banco de dados cliente-servidor orientados a objeto, em que se espera que as transações sejam relativamente longas. Indique dois motivos para os servidores de dados poderem ser populares para bancos de dados orientados a objeto, mas não para bancos de dados relacionais.

20.8 O que é desescalada de bloqueio, e sob quais condições ela é exigida? Por que ela não é exigida se a unidade de envio de dados for um item?

20.9 Suponha que você estivesse encarregado das operações de banco de dados de uma empresa cuja tarefa principal é processar transações. Suponha que a empresa esteja crescendo rapidamente a cada ano, e tenha crescido além do seu sistema de computador atual. Quando você estiver escolhendo um novo computador paralelo, que medida será mais relevante – ganho de velocidade, ganho de escala em batch ou ganho de escala de transação? Por quê?

20.10 Os sistemas de banco de dados normalmente são implementados como um conjunto de processos (ou threads) compartilhando uma área de memória compartilhada.

a. Como é controlado o acesso à área de memória compartilhada?

b. O bloqueio em duas fases é apropriado para colocar em série o acesso às estruturas de dados na memória compartilhada? Explique sua resposta.

20.11 Quais são os fatores que podem atuar contra o ganho de escala linear em um sistema de processamento de transações? Quais dos fatores provavelmente serão os mais importantes em cada uma das seguintes arquiteturas: memória compartilhada, disco compartilhado ou nada compartilhado?

20.12 As velocidades do processador cresceram muito mais rápido do que as velocidades de acesso à memória. Que impacto isso tem sobre o número de processadores que efetivamente podem compartilhar uma memória comum?

20.13 Considere um banco que tenha uma coleção de sites, cada um executando um sistema de banco de dados. Suponha que a única maneira de os bancos de dados interagirem é por transferência eletrônica de dinheiro entre eles. Esse tipo de sistema se qualificaria como um banco de dados distribuído? Por quê?

Notas bibliográficas

Hennessy *et al.* [2002] oferecem uma introdução excelente à área de arquitetura de computador.

Gray e Reuter [1993] oferecem uma descrição em forma de livro-texto sobre processamento de transação, incluindo a arquitetura de sistemas cliente-servidor e distribuídos. As notas bibliográficas do Capítulo 4 oferecem referências a mais informações sobre ODBC, JDBC e outras APIs de acesso a banco de dados.

Carey *et al.* [1991] e Franklin *et al.* [1993] descrevem as técnicas de caching de dados para sistemas de banco de da-

dos cliente-servidor. Biliris e Orenstein [1994] retratam os sistemas de gerenciamento de armazenamento de objeto, incluindo questões relacionadas a cliente-servidor. Franklin et al. [1992] e Mohan e Narang [1994] descrevem as técnicas de recuperação para sistemas cliente-servidor.

DeWitt e Gray [1992] explicam os sistemas de banco de dados paralelos, incluindo sua arquitetura e medidas de desempenho. Um estudo sobre arquiteturas de computador paralelas é apresentado por Duncan [1990]. Dubois e Thakkar [1992] oferecem uma coleção de artigos sobre arquiteturas escaláveis de memória compartilhada. Clusters DEC rodando o Rdb foram um dos primeiros usuários comerciais da arquitetura de banco de dados de disco compartilhado. Rdb agora pertence à Oracle e se chama Oracle Rdb. A Digital Equipment Corporation (DEC) agora pertence à Compaq, que, por sua vez, foi incorporada pela HP. A máquina de banco de dados Teradata esteve entre os primeiros sistemas comerciais a usar a arquitetura de banco de dados nada compartilhado. Os protótipos de pesquisa Grace e Gamma também usavam arquiteturas do tipo nada compartilhado.

Ozsu e Valduriez [1999] oferece um livro-texto dos sistemas de banco de dados distribuídos. Outras referências a sistemas de banco de dados paralelos e distribuídos aparecem nas notas bibliográficas dos Capítulos 21 e 22, respectivamente.

Comer e Droms [2003] e Thomas [1996] descrevem as redes de computadores e a Internet. Tanenbaum [2002] e Halsall [1996] oferecem introduções gerais sobre redes de computadores.

CAPÍTULO 21

Bancos de dados paralelos

Neste capítulo, discutimos os algoritmos fundamentais para os sistemas de banco de dados paralelos que são baseados no modelo de dados relacional. Em particular, focalizamos a colocação dos dados em vários discos e a avaliação paralela das operações relacionais, ambos instrumentos no sucesso dos bancos de dados paralelos.

Introdução

Há 15 anos, os sistemas de banco de dados paralelos quase foram abandonados, até mesmo por alguns de seus defensores mais leais. Hoje, eles são comercializados com sucesso por praticamente todo fornecedor de sistema de banco de dados. Várias tendências alimentaram essa transição:

- Os requisitos de transação das organizações cresceram com o uso cada vez maior dos computadores. Além do mais, o crescimento da World Wide Web criou muitos sites com milhões de visitantes, e as quantidades de dados cada vez maiores, coletadas por esses visitantes, produziram bancos de dados extremamente grandes em muitas empresas.
- As organizações estão usando esses volumes de dados cada vez maiores – como dados sobre quais itens as pessoas compram, em quais links da Web os usuários clicam, e quando as pessoas fazem ligações telefônicas – para planejar suas atividades e preços. As consultas usadas para tais finalidades são denominadas **consultas de apoio à decisão**, e os requisitos de dados para tais consultas podem chegar a terabytes. Sistemas de único processador não são capazes de lidar com volumes de dados tão grandes nas velocidades exigidas.
- A natureza orientada a conjunto das consultas de banco de dados é naturalmente adequada ao paralelismo. Diversos sistemas comerciais e de pesquisa demonstraram o poder e a escalabilidade do processamento paralelo da consulta.
- À medida que os microprocessadores se tornaram baratos, máquinas paralelas se tornaram comuns e relativamente baratas.

Conforme discutimos no Capítulo 20, o paralelismo é usado para oferecer ganho de velocidade, em que as consultas são executadas mais rapidamente porque são oferecidos mais recursos, como processadores e discos. O paralelismo também é usado para oferecer ganho de escala, em que cargas de trabalho cada vez maiores são tratadas sem aumentar o tempo de resposta, por meio de um aumento no grau de paralelismo.

Esboçamos, no Capítulo 20, as diferentes arquiteturas para sistemas de banco de dados paralelos: memória compartilhada, disco compartilhado, nada compartilhado e arquiteturas hierárquicas. Resumindo, nas arquiteturas de memória compartilhada, todos os processadores compartilham uma memória comum e discos; nas arquiteturas de disco compartilhado, os processadores possuem memórias independentes, mas compartilham discos; nas as arquiteturas de nada compartilhado, os processadores não compartilham nem memória nem discos; e as arquiteturas hierárquicas possuem nós que não compartilham nem memória nem discos entre si, mas internamente cada nó possui uma memória compartilhada ou uma arquitetura de disco compartilhado.

Paralelismo de E/S

Em sua forma mais simples, o **paralelismo de E/S** refere-se à redução de tempo exigida para apanhar relações do disco,

particionando as relações sobre múltiplos discos. A forma mais comum de particionamento de dados em um ambiente de banco de dados paralelo é o *particionamento horizontal*. No **particionamento horizontal**, as tuplas de uma relação são divididas (ou desagrupadas) entre muitos discos, de modo que cada tupla reside em um disco. Várias estratégias de particionamento foram propostas.

Técnicas de particionamento

Apresentamos três estratégias básicas de particionamento de dados. Considere que existem n discos, $D_0, D_1, \ldots, D_{n-1}$, pelos quais os dados devem ser particionados.

- **Rodízio**. Essa estratégia varre a relação em qualquer ordem e envia a i-ésima tupla ao disco número $D_{i \bmod n}$. O esquema de rodízio garante uma distribuição uniforme de tuplas pelos discos; ou seja, cada disco tem aproximadamente o mesmo número de tuplas dos outros.
- **Particionamento de hash**. Essa estratégia de desagrupamento designa um ou mais atributos do esquema de determinada relação como atributos de particionamento. Uma função de hash é escolhida, cujo intervalo é $\{0, 1, \ldots, n-1\}$. Cada tupla da relação original tem o hash calculado sobre os atributos de particionamento. Se a função de hash retornar i, então a tupla é colocada no disco D_i.
- **Particionamento de intervalo**. Essa estratégia distribui intervalos contíguos de valor de atributo a cada disco. Ela escolhe um atributo de particionamento, A, como **vetor de particionamento**. A relação é particionada da seguinte forma. Considere que $[v_0, v_1, \ldots, v_{n-2}]$ indica o vetor de particionamento, de modo que, se $i < j$, então $v_i < v_j$. Considere uma tupla t tal que $t[A] = x$. Se $x < v_0$, então t vai para o disco D_0. Se $x \geq v_{n-2}$, então t vai para o disco D_{n-1}. Se $v_i \leq x < v_{i+1}$, então t vai para o disco D_{i+1}.

  Por exemplo, o particionamento de intervalo com três discos de número 0, 1 e 2 pode atribuir tuplas com valores menores que 5 ao disco 0, valores entre 5 e 40 ao disco 1 e valores maiores que 40 ao disco 2.

Comparação de técnicas de particionamento

Quando uma relação tiver sido particionada entre vários discos, podemos recuperá-la em paralelo, usando todos os discos. De modo semelhante, quando uma relação está sendo particionada, ela pode ser escrita em vários discos em paralelo. Assim, as taxas de transferência para leitura ou escrita de uma relação inteira são muito mais rápidas com o paralelismo de E/S do que sem ele. Porém, ler uma relação inteira, ou *varrer uma relação*, é somente um tipo de acesso aos dados. O acesso aos dados pode ser classificado da seguinte maneira:

1. Varrendo a relação inteira
2. Localizando uma tupla de forma associativa (por exemplo, *nome_funcionário* = *"Campbell"*); essas consultas, chamadas consultas pontuais, buscam tuplas que possuem um valor especificado para um atributo especificado
3. Localizando todas as tuplas para as quais o valor de determinado atributo se encontra dentro de um intervalo especificado (por exemplo, 10.000 < *salário* < 20.000); essas consultas são denominadas **consultas de intervalo**.

As diferentes técnicas de particionamento admitem esses tipos de acesso em diferentes níveis de eficiência:

- **Rodízio**. O esquema é adequado de forma ideal para aplicações que querem ler a relação inteira seqüencialmente para cada consulta. Com esse esquema, consultas pontuais e consultas de intervalo são complicadas de se processar, pois cada um dos n discos precisa ser usado para a busca.
- **Particionamento por hash**. Esse esquema é mais adequado para consultas pontuais baseadas no atributo de particionamento. Por exemplo, se uma relação for particionada no atributo *número_telefone*, então podemos responder à consulta "Localizar o registro do empregado com *número_telefone* = 555-3333" aplicando a função de hash do particionamento a 555-3333 e depois pesquisando nesse disco. Direcionar uma consulta a um único disco economiza o custo de partida de iniciar uma consulta sobre vários discos e deixa os outros discos livres para processar outras consultas.

  O particionamento por hash também é útil para varreduras seqüenciais da relação inteira. Se uma função de hash for uma boa função de otimização, e os atributos de particionamento formarem uma chave da relação, então o número de tuplas em cada um dos discos é aproximadamente o mesmo, sem muita variância. Logo, o tempo gasto para varrer a relação é de cerca de $1/n$ do tempo exigido para varrer a relação em um único sistema de disco.

  Contudo, o esquema não é muito bem adequado para consultas pontuais sobre atributos não de particionamento. O particionamento baseado em hash também não é muito adequado para responder a consultas de intervalo porque, normalmente, as funções de hash não preservam a proximidade dentro do intervalo. Portanto, todos os discos precisam ser varridos para as consultas de intervalo serem respondidas.
- **Particionamento de intervalo**. Esse esquema é bem adequado para consultas pontuais e de intervalo sobre o atributo de particionamento. Para consultas pontuais,

podemos consultar o vetor de particionamento a fim de localizar o disco em que a tupla reside. Para consultas de intervalo, consultamos o vetor de particionamento para encontrar o intervalo de discos em que as tuplas podem residir. Nos dois casos, a busca se estreita a exatamente aqueles discos que poderiam ter tuplas de interesse.

Uma vantagem desse recurso é que, se houver apenas algumas tuplas no intervalo consultado, então a consulta normalmente é enviada a um disco, em vez de todos os discos. Como outros discos podem ser usados para responder a outras consultas, o particionamento de intervalo resulta em maior throughput enquanto mantem um bom tempo de resposta. Por outro lado, se houver muitas tuplas no intervalo consultado (como existem quando o intervalo consultado é uma fração maior do domínio da relação), muitas tuplas precisam ser recuperadas de poucos discos, resultando em um gargalo de E/S (hot spot) nesses discos. Neste exemplo de **distorção de execução**, todo o processamento ocorre em uma partição – ou apenas algumas. Ao contrário, o particionamento de hash e o de rodízio usariam todos os discos para tais consultas, ocasionando um tempo de resposta mais rápido para aproximadamente o mesmo throughput.

O tipo de particionamento também afeta outras operações relacionais, como junções, como veremos na seção "Paralelismo intra-operação". Assim, a escolha de técnica de particionamento também depende das operações que precisam ser executadas. Em geral, o particionamento de hash ou o particionamento de intervalo são preferíveis ao particionamento de rodízio.

Em um sistema com muitos discos, o número de discos pelos quais uma relação é particionada pode ser escolhido desta maneira: se uma relação tiver apenas algumas tuplas que caberão em um único bloco de disco, então é melhor atribuir a relação a um único disco. Relações grandes são preferivelmente particionadas por todos os discos disponíveis. Se uma relação consistir em m blocos de disco e houver n discos disponíveis no sistema, então a relação deverá receber a alocação de $\min(m, n)$ discos.

Tratamento da distorção

Quando uma relação é particionada (por uma técnica diferente do rodízio), pode haver uma **distorção** na distribuição de tuplas, com uma alta porcentagem de tuplas colocadas em algumas partições e menos tuplas em outras partições. As maneiras como a distorção pode aparecer são classificadas como:

- Distorção de valor de atributo
- Distorção de partição

A **distorção de valor de atributo** refere-se ao fato de que alguns valores aparecem nos atributos de particionamento de muitas tuplas. Todas as tuplas com o mesmo valor para o atributo de particionamento acabam na mesma partição, resultando na distorção. A **distorção de partição** refere-se ao fato de que pode haver desequilíbrio de carga no particionamento, mesmo quando não existe distorção de atributo.

A distorção de valor de atributo pode resultar em particionamento distorcido, independente de se o particionamento de intervalo ou o particionamento de hash é utilizado. Se o vetor de partição não for escolhido com cuidado, o particionamento de intervalo pode resultar em distorção de partição. A distorção de partição é menos provável com o particionamento de hash, se uma boa função de hash for escolhida.

Como vimos na seção "Ganho de velocidade e escala" do Capítulo 20, até mesmo uma pequena distorção pode resultar em uma diminuição significativa no desempenho. A distorção se torna um problema maior com um grau maior de paralelismo. Por exemplo, se uma relação de 1000 tuplas for dividida em 10 partes, e a divisão for distorcida, então poderá haver algumas partições com tamanho menor que 100 e algumas partições com tamanho maior que 100; se até mesmo uma partição tiver o tamanho 200, o ganho de velocidade que obteríamos acessando as partições em paralelo é de apenas 5, em vez dos 10 pelos quais esperaríamos. Se a mesma relação tiver de ser particionada em 100 partes, uma partição terá 10 tuplas na média. Se até mesmo uma partição tiver 40 tuplas (o que é possível, dada a grande quantidade de partições), o ganho de velocidade que obteríamos acessando todas em paralelo seria de 25, em vez de 100. Assim, vemos que a perda de ganho de velocidade devido à distorção aumenta com o paralelismo.

Um **vetor de particionamento de intervalo balanceado** pode ser construído pela classificação: primeiro a relação é classificada sobre os atributos de particionamento; depois é varrida na ordem classificada. Depois que cada $1/n$ da relação tiver sido lido, o valor do atributo de particionamento da próxima tupla é somado ao vetor de partição. Aqui, n indica o número de partições a serem construídas. Caso haja muitas tuplas com o mesmo valor para o atributo de particionamento, a técnica ainda pode resultar em alguma distorção. A principal desvantagem desse método é a sobrecarga de E/S extra advinda da classificação inicial.

A sobrecarga de E/S para a construção de vetores de partição de intervalo balanceados pode ser reduzida pela construção e armazenamento de uma tabela de freqüência, ou **histograma**, dos valores de atributo para cada transação de cada relação. A Figura 21.1 mostra um exemplo de um histograma para um atributo de valor inteiro que apanha valores no intervalo de 1 a 25. Um histograma ocupa apenas um pequeno espaço, de modo que os histogramas sobre diver-

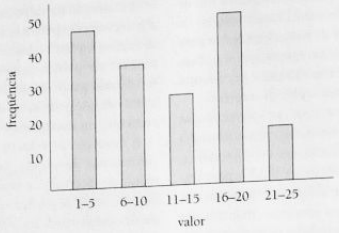


Figura 21.1 Exemplo de histograma.

sos atributos diferentes podem ser armazenados no catálogo do sistema. É muito simples construir uma função de particionamento de intervalo balanceado dado um histograma sobre os atributos de particionamento. Se o histograma não estiver armazenado, ele pode ser calculado aproximadamente pela amostragem da relação, usando apenas tuplas de um subconjunto escolhido aleatoriamente dos blocos de disco da relação.

Outra técnica para minimizar o efeito da distorção, principalmente com o particionamento de intervalo, é usar *processadores virtuais*. Na técnica de **processador virtual**, fingimos que existem várias vezes tantos *processadores virtuais* quanto o número de processadores reais. Qualquer uma das técnicas de particionamento e técnicas de avaliação de consulta que estudaremos mais adiante neste capítulo poderão ser usadas, mas elas mapeiam tuplas e trabalham para processadores virtuais no lugar de processadores reais. Os processadores virtuais, por sua vez, são mapeados para processadores reais, normalmente pelo particionamento de rodízio.

A idéia é que, mesmo que um intervalo tivesse muito mais tuplas do que os outros devido à distorção, essas tuplas seriam divididas por vários intervalos de processadores virtuais. A alocação de rodízio dos processadores virtuais para processadores reais distribuiria o trabalho extra entre vários processadores reais, de modo que um processador não tenha de agüentar todo o peso do trabalho.

Paralelismo interconsulta

No **paralelismo interconsulta**, diferentes consultas ou transações são executadas em paralelo umas com as outras. O throughtput da transação pode ser aumentado por essa forma de paralelismo. Porém, os tempos de resposta das transações individuais não são mais rápidos do que seriam se as transações fossem executadas isoladamente. Assim, o principal uso do paralelismo interconsulta é obter ganho de escala em um sistema de processamento de transações, para aceitar uma quantidade maior de transações por segundo.

O paralelismo interconsulta é a forma mais fácil de paralelismo para dar suporte em um sistema de banco de dados – particularmente em um sistema paralelo de memória compartilhada. Os sistemas de banco de dados projetados para sistemas de único processador podem ser usados com poucas ou nenhuma mudança em uma arquitetura paralela de memória compartilhada, pois até mesmo os sistemas de banco de dados seqüenciais admitem o processamento concorrente. As transações que teriam operado de uma maneira concorrente em tempo compartilhado numa máquina seqüencial operam em paralelo na arquitetura paralela de memória compartilhada.

O suporte ao paralelismo interconsulta é mais complicado em uma arquitetura de disco compartilhado ou nada compartilhado. Os processadores precisam realizar algumas tarefas, como bloqueio e logging, de uma maneira coordenada, e isso requer que eles passem mensagens entre si. Um sistema de banco de dados paralelo também precisa garantir que dois processadores não atualizem os mesmos dados independentemente ao mesmo tempo. Além disso, quando um processador acessa ou atualiza dados, o sistema de banco de dados precisa garantir que o processador tenha a versão mais recente dos dados em seu pool de buffer. O problema de garantir que a versão seja a mais recente é conhecido como problema de **coerência de cache**.

Diversos protocolos estão disponíveis para garantir a coerência de cache; normalmente, os protocolos de coerência de cache são integrados a protocolos de controle de concorrência, de modo que sua sobrecarga é reduzida. Um protocolo desse tipo para um sistema de disco compartilhado é este:

1. Antes de qualquer acesso para leitura ou escrita a uma página, uma transação bloqueia a página no

modo compartilhado ou exclusivo, conforme a necessidade. Imediatamente após a transação obter um bloqueio compartilhado ou exclusivo sobre uma página, ela também lê a cópia mais recente da página do disco compartilhado.

2. Antes que uma transação libere um bloqueio exclusivo sobre uma página, ela esvazia a página para o disco compartilhado; depois, ela libera o bloqueio.

Esse protocolo garante que, quando uma transação define um bloqueio compartilhado ou exclusivo sobre uma página, ela receba a cópia correta da página.

Protocolos mais complexos evitam a leitura repetida e a escrita em disco exigida pelo protocolo anterior. Esses protocolos não escrevem páginas no disco quando os bloqueios exclusivos são liberados. Quando um bloqueio compartilhado ou exclusivo é obtido, se a versão mais recente de uma página estiver no pool de buffer de algum processador, a página é obtida de lá. Os protocolos precisam ser projetados para lidar com solicitações concorrentes. Os protocolos de disco compartilhado podem ser estendidos para arquiteturas nada compartilhado por meio deste esquema: cada página possui um **processador home** P_i e é armazenada no disco D_i. Quando outros processadores querem ler ou escrever a página, enviam solicitações ao processador home P_i da página, pois não podem se comunicar diretamente com o disco. As outras ações são iguais às dos protocolos de disco compartilhado.

Os sistemas Oracle e Oracle Rdb são exemplos de sistemas de banco de dados paralelos de disco compartilhado que admitem paralelismo interconsulta.

Paralelismo intraconsulta

O **paralelismo intraconsulta** refere-se à execução de uma única consulta em paralelo em vários processadores e discos. O uso do paralelismo intraconsulta é importante para agilizar consultas de longa duração. O paralelismo interconsulta não ajuda nessa tarefa, pois cada consulta é executada seqüencialmente.

Para ilustrar a avaliação paralela de uma consulta, considere uma consulta que exige que uma relação seja classificada. Suponha que a relação tenha sido particionada entre vários discos pelo intervalo particionado sobre algum atributo, e a classificação seja solicitada sobre o atributo do particionamento. A operação de classificação pode ser implementada pela classificação de cada partição em paralelo, depois concatenando as partições classificadas para chegar a relação classificada final.

Assim, podemos colocar uma consulta em paralelo pela paralelização de operações individuais. Existe outra fonte de paralelismo na avaliação de uma consulta: a *árvore de operadores* para uma consulta pode conter várias operações. Podemos paralelizar a avaliação da árvore de operadores avaliando em paralelo algumas das operações que não dependem uma da outra. Além do mais, como o Capítulo 13 menciona, podemos ser capazes de canalizar a saída de uma operação a outra. As duas operações podem ser executadas em paralelo sobre processadores separados, um gerando saída que é consumida pelo outro, até mesmo enquanto ela está sendo gerada.

Resumindo, a execução de uma única consulta pode ser colocada em paralelo de duas maneiras:

- **Paralelismo intra-operação.** Podemos agilizar o processamento de uma consulta colocando em paralelo a execução de cada operação individual, como sort, select, project e join. Consideramos o paralelismo intra-operação na seção anterior.
- **Paralelismo interoperação.** Podemos agilizar o processamento de uma consulta executando em paralelo as diferentes operações em uma expressão de consulta. Consideramos essa forma de paralelismo na seção "Paralelismos interoperações".

As duas formas de paralelismo são complementares e podem ser usadas simultaneamente em uma consulta. Como o número de operações em uma consulta típica é pequeno, em comparação com o número de tuplas processadas por cada operação, a primeira forma de paralelismo pode se expandir melhor com o paralelismo aumentado. Porém, com o número relativamente pequeno de processadores nos sistemas paralelos típicos de hoje, as duas formas de paralelismo são importantes.

Na discussão a seguir, sobre paralelismo de consultas, consideramos que as consultas são **apenas de leitura**. A escolha de algoritmos para colocar em paralelo a avaliação de consulta depende da arquitetura da máquina. Em vez de apresentar algoritmos para cada arquitetura separadamente, usamos um modelo de arquitetura nada compartilhado em nossa descrição. Assim, descrevemos explicitamente quando os dados precisam ser transferidos de um processador para outro. Podemos simular esse modelo facilmente usando as outras arquiteturas, pois a transferência de dados pode ser feita por meio da memória compartilhada em uma arquitetura de memória compartilhada, e por meio de discos compartilhados em uma arquitetura de disco compartilhado. Logo, os algoritmos para arquiteturas nada compartilhado também podem ser usados nas outras arquiteturas. Mencionamos ocasionalmente como os algoritmos podem ser otimizados ainda mais para sistemas de memória compartilhada ou disco compartilhado.

Para simplificar a apresentação dos algoritmos, consideramos que existem n processadores, $P_0, P_1, \ldots, P_{n-1}$, e n dis-

cos $D_0, D_1, \ldots, D_{n-1}$, em que o disco D_i está associado ao processador P_i. Um sistema real pode ter vários discos por processador. Não é difícil estender os algoritmos para permitir vários discos por processador: simplesmente permitimos que D_i seja um conjunto de discos. Porém, por simplicidade, consideramos aqui que D_i é um único disco.

Paralelismo intra-operação

Como operações relacionais atuam sobre relações contendo grandes conjuntos de tuplas, podemos colocar em paralelo as operações executando-as em paralelo em diferentes subconjuntos das relações. Como o número de tuplas em uma relação pode ser grande, o grau de paralelismo é potencialmente enorme. Assim, o paralelismo intra-operação é natural em um sistema de banco de dados. Vamos estudar as versões paralelas de algumas operações relacionais comuns nas seções "Classificação paralela" a "Outras operações relacionais".

Classificação paralela

Suponha que queiramos classificar uma relação que reside em n discos $D_0, D_1, \ldots, D_{n-1}$. Se a relação tiver sido particionada por intervalo sobre os atributos nos quais deve ser classificada, então, como observamos na seção "Comparação de técnicas de particionamento", podemos classificar cada partição separadamente, além de concatenar os resultados para obter a relação classificada inteira. Como as tuplas são particionadas sobre *n* discos, o tempo exigido para a leitura da relação inteira é reduzido pelo acesso paralelo.

Se a relação tiver sido particionada de qualquer outra maneira, podemos classificá-la de uma destas formas:

1. Podemos particioná-la por intervalo sobre os atributos de classificação e depois classificar cada partição separadamente.
2. Podemos usar uma versão paralela do algoritmo sort-merge externo.

Classificação por particionamento de intervalo

A **classificação por particionamento de intervalo** funciona em duas etapas: primeiro, particionando os intervalos da relação, depois classificando cada partição separadamente. Quando classificamos por particionamento de intervalo da relação, não é necessário particionar a relação por intervalo sobre o mesmo conjunto de processadores ou discos em que essa relação está armazenada. Suponha que escolhemos os processadores $P_0, P_1, \ldots, P_m$, onde $m < n$, para classificar a relação. Existem duas etapas envolvidas nessa operação:

1. Redistribuir as tuplas na relação, usando uma estratégia de partição de intervalo, de modo que todas as tuplas que se encontram dentro do *i*-ésimo intervalo sejam enviadas ao processador P_i, que armazena a relação temporariamente no disco D_i.
   Para implementar o particionamento de intervalo, em paralelo, cada processador lê as tuplas do seu disco e as envia para o seu processador de destino. Cada processador $P_0, P_1, \ldots, P_m$ também recebe tuplas pertencentes à sua partição e as armazena localmente. Essa etapa requer sobrecarga de E/S de disco e comunicação.
2. Cada um dos processadores classifica sua partição da relação localmente, sem interação com os outros processadores. Cada processador executa a mesma operação – a saber, classificação – sobre um conjunto de dados diferente. (A execução da mesma operação em paralelo sobre diferentes conjuntos de dados é chamada **paralelismo de dados**.)
   A operação de mesclagem final é trivial, pois o particionamento de intervalo na primeira fase garante que, para $1 \le i < j \le m$, os valores de chave no processador P_i são todos menores que os valores de chave em P_j.

Temos de realizar o particionamento de intervalo com um bom vetor de particionamento de intervalo, de modo que cada partição tenha aproximadamente o mesmo número de tuplas. O particionamento de processador virtual também pode ser usado para reduzir a distorção.

Sort-Merge externo paralelo

O **sort-merge externo paralelo** é uma alternativa ao particionamento de intervalo. Suponha que uma relação já tenha sido particionada entre os discos $D_0, D_1, \ldots, D_{n-1}$ (não importa como a relação foi particionada). O sort-merge externo paralelo funciona desta maneira:

1. Cada processador P_i classifica os dados localmente no disco D_i.
2. O sistema, então, mescla as rodadas classificadas em cada processador para chegar à saída classificada final.

A mesclagem das rodadas classificadas na etapa 2 pode ser colocada em paralelo por esta seqüência de ações:

1. O sistema particiona por intervalo as partições classificadas em cada processador P_i (todas pelo mesmo vetor de partição) com a ajuda dos processadores $P_0, P_1, \ldots, P_{m-1}$. Ele envia as tuplas em ordem, de

modo que cada processador receba as tuplas nos fluxos classificados.

2. Cada processador P_i realiza uma mesclagem sobre os fluxos enquanto eles são recebidos, para obter uma única rodada classificada.
3. O sistema concatena as rodadas classificadas nos processadores $P_0, P_1, \ldots, P_{m-1}$ para chegar ao resultado final.

Conforme descrevemos, essa seqüência de ações resulta em uma forma interessante de **distorção de execução**, pois a princípio cada processador envia todos os blocos da partição 0 para P_0, depois cada processador envia todos os blocos da partição 1 para P_1, e assim por diante. Assim, enquanto o envio acontece em paralelo, as tuplas receptoras se tornam seqüenciais: primeiro, apenas P_0 recebe tuplas, depois, apenas P_1 recebe tuplas, e assim por diante. Para evitar esse problema, cada processador envia repetidamente um bloco de dados para cada partição. Em outras palavras, cada processador envia o primeiro bloco de cada partição, depois envia o segundo bloco de cada partição etc. Como resultado, todos os processadores recebem dados em paralelo.

Algumas máquinas, como as máquinas da série Teradata DBC, utilizam hardware especializado para realizar a mesclagem. A rede de interconexão Y-net nas máquinas Teradata DBC pode mesclar a saída de vários processadores para gerar uma única saída classificada.

Junção paralela

A operação de junção requer que o sistema teste pares de tuplas para ver se eles satisfazem a condição de junção; se satisfizerem, o sistema acrescenta o par na saída da junção. Os algoritmos de junção paralela tentam dividir os pares a serem testados por vários processadores. Cada processador, então, calcula parte da junção localmente. Depois, o sistema coleta os resultados de cada processador para produzir o resultado final.

Junção particionada

Para certos tipos de junções, como as equijunções e junções naturais, é possível particionar as duas relações de entrada por intermédio dos processadores e calcular a junção localmente em cada processador. Suponha que estejamos usando n processadores e que as relações a serem juntadas sejam r e s. A junção particionada, então, funciona desta maneira: o sistema particiona as relações r e s em n partições cada uma, indicadas por $r_0, r_1, \ldots, r_{n-1}$ e $s_0, s_1, \ldots, s_{n-1}$. O sistema envia partições r_i e s_i ao processador P_i, onde sua junção é calculada localmente.

A técnica de junção particionada funciona corretamente se a junção for uma equijunção (por exemplo, $r \bowtie_{r.A=s.B} s$) e se particionarmos r e s pela mesma função de particionamento sobre seus atributos de junção. A idéia de particionamento é exatamente a mesma daquela por trás da etapa de particionamento da junção de hash. Porém, em uma junção particionada, existem duas maneiras diferentes de particionar r e s:

- Particionamento de intervalo sobre os atributos de junção
- Particionamento de hash sobre os atributos de junção

De qualquer forma, a mesma função de particionamento precisa ser usada para as duas relações. Para o particionamento de intervalo, o mesmo vetor de partição precisa ser usado para as duas relações. Para o particionamento de hash, a mesma função de hash precisa ser usada nas duas relações. A Figura 21.2 representa o particionamento em uma junção paralela particionada.

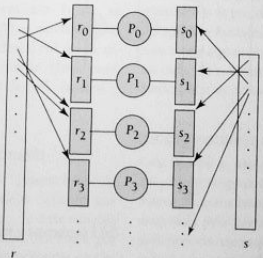


Figura 21.2 Junção paralela particionada.

Quando as relações são particionadas, podemos usar qualquer técnica de junção localmente em cada processador P_i para calcular a junção de r_i e s_i. Por exemplo, a junção de hash, a junção de mesclagem ou a junção de loop aninhado poderia ser usada. Assim, podemos usar o particionamento para colocar em paralelo qualquer técnica de junção.

Se uma ou ambas as relações r e s já estiverem particionadas sobre os atributos da junção (por particionamento de hash ou particionamento de intervalo), o trabalho necessário para o particionamento é extremamente reduzido. Se as relações não forem particionadas, ou se forem particionadas sobre atributos diferentes dos atributos de junção, então as tuplas precisam ser reparticionadas. Cada processador P_i lê as tuplas no disco D_i, calcula para cada tupla t a partição j à qual t pertence e envia a tupla t ao processador P_j. O processador P_j armazena as tuplas no disco D_j.

Podemos otimizar o algoritmo de junção usado localmente em cada processador para reduzir a E/S colocando em buffer algumas das tuplas para a memória, em vez de gravá-las em disco. Descrevemos essas otimizações na seção "Junção de hash paralelo particionado".

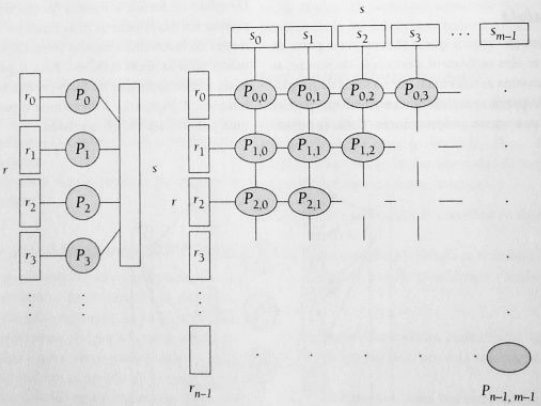
A distorção apresenta um problema especial quando o particionamento de intervalo é utilizado, pois um vetor de partição que divide uma relação da junção em partições de mesmo tamanho pode dividir as outras relações em partições de tamanhos bastante variados. O vetor de partição deverá ser tal que $|r_i| + |s_i|$ (ou seja, a soma dos tamanhos de r_i e s_i) seja aproximadamente igual por todo o $i = 0, 1, \ldots, n - 1$. Com uma boa função de hash, o particionamento de hash provavelmente terá uma distorção menor, exceto quando existem muitas tuplas com os mesmos valores para os atributos da junção.

Junção fragmentar-e-replicar

O particionamento não se aplica a todos os tipos de junções. Por exemplo, se a condição de junção for uma desigualdade, como $r \bowtie_{r.a < s.b} s$, é possível que todas as tuplas na junção r se juntem com alguma tupla em s (e vice-versa). Assim, pode não haver um modo fácil de particionar r e s de modo que as tuplas na partição r_i se juntem apenas com tuplas na partição s_i.

Podemos colocar essas associações em paralelo usando uma técnica chamada fragmentar-e-replicar. Primeiro, consideramos um caso especial de fragmentar-e-replicar – **junção fragmentar-e-replicar assimétrica** –, que funciona da seguinte maneira:

1. O sistema particiona uma as relações – digamos, r. Qualquer técnica de particionamento pode ser usada sobre r, incluindo o particionamento de rodízio.



(a) Fragmentar e replicar assimétrico

(b) Fragmentar e replicar

Figura 21.3 Esquemas fragmentar-e-replicar.

2. O sistema replica a outra relação, s, por todos os processadores.
3. O processador P_i então, calcula no local a junção de r_i com todos os s, usando qualquer técnica de junção.

O esquema fragmentar-e-replicar assimétrico aparece na Figura 21.3a. Se r já estiver armazenada pelo particionamento, não é preciso particioná-la mais na etapa 1. Tudo o que é preciso é replicar s por todos os processadores.

O caso geral da **junção fragmentar-e-replicar** aparece na Figura 21.3b; ele funciona desta forma: o sistema particiona a relação r em n partições, $r_0, r_1, \ldots, r_{n-1}$, e particiona s em m partições, $s_0, s_1, \ldots, s_{m-1}$. Como antes, qualquer técnica de particionamento pode ser usada sobre r e sobre s. Os valores de m e n não precisam ser iguais, mas eles precisam ser escolhidos de modo que existam pelo menos $m * n$ processadores. A técnica fragmentar-e-replicar assimétrica é simplesmente um caso especial de fragmentar-e-replicar, onde $m = 1$. Fragmentar-e-replicar reduz os tamanhos das relações em cada processador, em comparação com o fragmentar-e-replicar assimétrico.

Considere que os processadores sejam $P_{0,0}, P_{0,1}, \ldots, P_{0,m-1}, P_{1,0}, \ldots, P_{n-1,m-1}$. O processador $P_{i,j}$ calcula a junção de r_i com s_j. Cada processador precisa apanhar as tuplas nas partições em que trabalha. Para fazer isso, o sistema replica r_i para os processadores $P_{i,0}, Pi,1, \ldots, P_{i,m-1}$ (que formam uma linha na Figura 21.3b), e replica s_i para os processadores $P_{0,i}, P_{1,i}, \ldots, P_{n-1,i}$ (que formam uma coluna na Figura 21.3b). Qualquer técnica de junção pode ser usada em cada processador $P_{i,j}$.

Fragmentar-e-replicar funciona com qualquer condição de junção, pois cada tupla em r pode ser testada com cada tupla em s. Assim, ela pode ser usada onde o particionamento não pode ser usado.

Fragmentar-e-replicar normalmente possui um custo mais alto do que o particionamento quando as duas relações são aproximadamente do mesmo tamanho, pois pelo menos uma das relações precisa ser replicada. Porém, se uma das relações – digamos, s – for pequena, pode ser mais barato replicar s por todos os processadores, em vez de reparticionar r e s sobre os atributos da junção. Nesse caso, fragmentar-e-replicar assimétrico é preferível, embora o particionamento possa ser utilizado.

Junção de hash paralelo particionado

A junção de hash particionado da seção "Junção hash" do Capítulo 13 pode ser colocada em paralelo. Suponha que tenhamos n processadores, $P_0, P_1, \ldots, P_{n-1}$, e duas relações r e s, de modo que as relações r e s sejam particionadas por múltiplos discos. Lembre-se, da seção mencionada do Capítulo 13, de que a relação menor é escolhida como relação de montagem. Se o tamanho de s for menor que o de r, o algoritmo de junção de hash paralelo prossegue da seguinte maneira:

1. Escolha uma função de hash – digamos, h_1 – que apanhe o valor do atributo de junção de cada tupla em r e s e mapeie a tupla para um dos n processadores. Considere que r_i indica as tuplas da relação r que são mapeadas para o processador P_i; de modo semelhante, considere que s_i indica as tuplas da relação s que são mapeadas para o processador P_i. Cada processador P_i lê as tuplas de s que estão em seu disco D_i e envia cada tupla ao processador apropriado com base na função de hash h_1.
2. À medida que o processador de destino P_i recebe as tuplas de s_i, ele as particiona ainda mais por outra função de hash, h_2, que o processador usa para calcular a junção de hash localmente. O particionamento nesse estágio é exatamente o mesmo que na fase de particionamento do algoritmo de junção de hash sequencial. Cada processador P_i executa essa etapa independentemente da de outros processadores.
3. Quando as tuplas de s tiverem sido distribuídas, o sistema redistribui a relação maior r pelos n processadores pela função de hash h_1, da mesma maneira como antes. Ao receber cada tupla, o processador de destino a reparticiona pela função h_2, assim como a relação de sonda é particionada no algoritmo de junção de hash sequencial.
4. Cada processador P_i executa as fases de montagem e sonda do algoritmo de junção de hash sobre as partições locais r_i e s_i de r e s para produzir uma partição do resultado final da junção de hash.

A junção de hash em cada processador é independente daquela de outros processadores, e receber as tuplas de r_i e s_i é semelhante a lê-las do disco. Portanto, qualquer uma das otimizações da junção de hash descritas no Capítulo 13 pode ser aplicada também ao caso paralelo. Em particular, podemos usar o algoritmo de junção de hash híbrido para colocar no cache da memória algumas das tuplas que chegam, evitando assim os custos de escrevê-las e lê-las de volta.

Junção paralela de loop aninhado

Para ilustrar o uso do paralelismo baseado em fragmentar-e-replicar, considere o caso em que a relação s é muito menor do que a relação r. Suponha que a relação r seja armazenada pelo particionamento; o atributo em que ela é particionada não importa. Suponha também que existe um índice sobre um atributo de junção da relação r em cada uma das partições da relação r.

Usamos a técnica fragmentar-e-replicar assimétrica, com a relação s sendo replicada e com o particionamento existente da relação r. Cada processador P_j em que uma partição da relação s é armazenada lê as tuplas da relação s armazenadas em D_j, e replica as tuplas para cada outro processador P_i. Ao final dessa fase, a relação s é replicada em todos os sites que armazenam tuplas da relação r.

Agora, cada processador P_i realiza uma junção de loop aninhado indexada da relação s com a i-ésima partição da relação r. Podemos superpor a junção de loop aninhado indexada com a distribuição de tuplas da relação s, para reduzir os custos da gravação das tuplas da relação s em disco e da leitura de volta. Porém, a replicação da relação s precisa ser sincronizada com a junção para que haja espaço suficiente nos buffers na memória em cada processador P_i de modo a manter as tuplas da relação s que foram recebidas, mas que ainda não foram usadas na junção.

Outras operações relacionais

A avaliação de outras operações relacionais também pode ser colocada em paralelo:

- **Seleção.** Considere a seleção $\sigma_\theta(r)$. Considere primeiro o caso em que θ tem a forma $a_i = v$, onde a_i é um atributo e v é um valor. Se a relação r for particionada sobre a_i, a seleção prossegue em um único processador. Se θ for da forma $l \leq a_i \leq u$ – ou seja, θ for uma seleção de intervalo – e a relação tiver sido particionada por intervalo sobre a_i, então a seleção prossegue em cada processador cuja partição superpõe com o intervalo de valores especificado. Em todos os outros casos, a seleção prossegue em paralelo em todos os processadores.
- **Eliminação de duplicatas.** As duplicatas podem ser eliminadas pela classificação; qualquer uma das técnicas de classificação paralelas pode ser usada, otimizada para eliminar duplicatas assim que aparecerem durante a classificação. Também podemos colocar a eliminação de duplicatas em paralelo particionando as tuplas (por particionamento de intervalo ou de hash) e eliminando duplicatas localmente em cada processador.
- **Projeção.** A projeção sem eliminação de duplicatas pode ser realizada à medida que as tuplas são lidas do disco em paralelo. Se as duplicatas tiverem de ser eliminadas, qualquer uma das técnicas descritas poderá ser usada.
- **Agregação.** Considere uma operação de agregação. Podemos colocar a operação em paralelo particionando a relação sobre os atributos de agrupamento, e depois calculando os valores agregados localmente em cada processador. Tanto o particionamento de hash quanto o particionamento de intervalo podem ser usados. Se a relação já estiver particionada sobre os atributos de agrupamento, a primeira etapa pode ser pulada.

Podemos reduzir o custo de transferência de tuplas durante o particionamento calculando parcialmente valores agregados antes do particionamento, pelo menos para as funções de agregação mais utilizadas. Considere uma operação de agregação sobre uma relação r, usando a função de agregação **sum** sobre o atributo B, com o agrupamento sobre o atributo A. O sistema pode realizar a operação em cada processador P_i sobre aquelas tuplas r armazenadas no disco D_i. Essa computação resulta em tuplas com somas parciais em cada processador; existe uma tupla em P_i para cada valor para o atributo A presente nas tuplas r armazenadas em D_i. O sistema particiona o resultado da agregação local sobre o atributo de agrupamento A e realiza a agregação novamente (sobre as tuplas com as somas parciais) em cada processador P_i para chegar ao resultado final.

Como resultado dessa otimização, menos tuplas precisam ser enviadas a outros processadores durante o particionamento. Essa idéia pode ser estendida com facilidade para as funções de agregação **min** e **max**. As extensões às funções de agregação **count** e **avg** ficam para o Exercício 21.10.

O paralelismo de outras operações é abordado em vários dos exercícios.

Custo da avaliação paralela das operações

Conseguimos o paralelismo particionando a E/S entre vários discos e particionando o trabalho da CPU entre múltiplos processadores. Se essa divisão for alcançada sem qualquer sobrecarga, e se não houver distorção na divisão do trabalho, uma operação paralela usando n processadores levará $1/n$ vezes o tempo da mesma operação sobre um único processador. Já sabemos como estimar o custo de uma operação como uma junção ou uma seleção. O custo de tempo do processamento paralelo seria, então, $1/n$ do custo de tempo do processamento sequencial da operação.

Também temos de levar em consideração os seguintes custos:

- **Custos de partida** para iniciar a operação em múltiplos processadores
- **Distorção** na distribuição de trabalho entre os processadores, com alguns processadores recebendo um número maior de tuplas que outros
- **Disputa por recursos** – como memória, disco e a rede de comunicação – resultando em atrasos
- **Custo de montagem** do resultado final transmitindo resultados parciais por cada processador

O tempo gasto por uma operação paralela pode ser estimado como

$$T_{part} + T_{mon} + \max(T_0, T_1, \ldots, T_{n-1})$$

onde Tpart é o tempo para particionar as relações, Tmon é o tempo para montagem dos resultados e T_i é o tempo gasto para a operação no processador P_i. Supondo que as tuplas sejam distribuídas sem qualquer distorção, o número de tuplas enviadas a cada processador pode ser estimado como $1/n$ do número total de tuplas. Ignorando a disputa, o custo T_i das operações em cada processador P_i pode então ser estimado pelas técnicas no Capítulo 13.

A estimativa anterior será uma estimativa otimista, pois a distorção é comum. Embora o desmembramento de uma única consulta em uma série de etapas paralelas reduza o tamanho da etapa média, é o tempo para o processamento da única etapa mais lenta que determina o tempo gasto para o processamento da consulta como um todo. Uma avaliação paralela particionada, por exemplo, é tão rápida quanto a mais lenta das execuções paralelas. Assim, qualquer distorção na distribuição do trabalho entre os processadores afeta bastante o desempenho.

O problema da distorção no particionamento está relacionado de perto ao problema do estouro de partição nas junções de hash seqüenciais (Capítulo 13). Podemos usar as técnicas de resolução e impedimento de estouro, desenvolvidas para junção de hash, para tratar da distorção quando o particionamento de hash for utilizado. Podemos usar o particionamento de intervalo balanceado e o particionamento de processador virtual para reduzir a distorção devida ao particionamento de intervalo, como na seção "Tratamento da distorção".

Paralelismo interoperações

Existem duas formas de paralelismo interoperações: paralelismo canalizado e paralelismo independente.

Paralelismo canalizado

Conforme discutimos no Capítulo 13, a canalização forma uma importante fonte de economia de computação para o processamento de consulta no banco de dados. Lembre-se de que, na canalização, as tuplas de saída de uma operação, *A*, são consumidas por uma segunda operação, *B*, mesmo antes que a primeira operação tenha produzido o conjunto inteiro de tuplas em sua saída. A principal vantagem da execução canalizada em uma avaliação seqüencial é que podemos executar uma seqüência dessas operações sem gravar qualquer um dos resultados intermediários em disco.

Sistemas paralelos utilizam a canalização principalmente pelo mesmo motivo que os sistemas seqüenciais. Porém, as canalizações também são uma fonte de paralelismo, da mesma forma como as canalizações de instrução são uma fonte de paralelismo no projeto do hardware. É possível executar operações *A* e *B* simultaneamente em diferentes processadores, de modo que *B* consuma tuplas em paralelo com *A* produzindo-as. Essa forma de paralelismo é chamada de **paralelismo canalizado**.

Considere uma junção de quatro relações:

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

Podemos estabelecer uma canalização que permita que as três junções sejam computadas em paralelo. Suponha que o processador P_1 receba a incumbência da computação de $temp_1 \leftarrow r_1 \bowtie r_2$, e P_2 esteja encarregado da computação de $r_3 \bowtie temp_1$. Enquanto P_1 trabalha com as tuplas em $r_1 \bowtie r_2$, ele torna essas tuplas disponíveis ao processador P_2. Assim, P_2 tem à sua disposição algumas das tuplas em $r_1 \bowtie r_2$ antes que P_1 tenha terminado seu trabalho. P_2 pode usar aquelas tuplas que estão disponíveis para iniciar a computação de $temp_1 \bowtie r_3$, mesmo antes que $r_1 \bowtie r_2$ seja totalmente computado por P_1. De modo semelhante, enquanto P_2 computa as tuplas em $(r_1 \bowtie r_2) \bowtie r_3$, ele torna essas tuplas disponíveis para P_3, que computa a junção dessas tuplas com r_4.

O paralelismo canalizado é útil com um pequeno número de processadores, mas não se expande muito bem. Primeiro, as cadeias de canalização geralmente não alcançam tamanho suficiente para oferecer um alto grau de paralelismo. Segundo, não é possível canalizar operadores relacionais que não produzem saída até que todas as entradas tenham sido acessadas, como a operação de diferença de conjunto. Terceiro, somente um pequeno ganho de velocidade é obtido para os casos freqüentes em que o custo de execução de um operador é muito maior que os dos outros.

Considerando todos os fatos, quando o grau de paralelismo é alto, a canalização é uma fonte de paralelismo menos importante do que o particionamento. O motivo real para o uso da canalização é que as execuções canalizadas podem evitar a gravação de resultados intermediários em disco.

Paralelismo independente

As operações em uma expressão de consulta que não dependem uma da outra podem ser executadas em paralelo. Essa forma de paralelismo é chamada **paralelismo independente**.

Considere a junção $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$. Logicamente, podemos computar $temp_1 \leftarrow r_1 \bowtie r_2$ em paralelo com $temp_2 \leftarrow r_3 \bowtie r_4$. Quando essas duas computações terminarem, computamos

$$temp_1 \bowtie temp_2$$

Para obter mais paralelismo, podemos canalizar as tuplas em $temp_1$ e $temp_2$ na computação de $temp_1 \bowtie temp_2$, que é executada por uma junção canalizada (seção "Algoritmos de avaliação para a canalização" do Capítulo 13).

Assim como o paralelismo canalizado, o paralelismo independente não oferece um alto grau de paralelismo e é menos útil em um sistema altamente paralelo, embora seja útil com um menor grau de paralelismo.

Otimização de consulta

Os otimizadores de consulta são responsáveis em grande parte pelo sucesso da tecnologia relacional. Lembre-se de que um otimizador de consulta apanha uma consulta e encontra o plano de execução mais barato entre os muitos planos de execução possíveis, que dão a mesma resposta.

Os otimizadores de consulta para a avaliação de consulta em paralelo são mais complicados do que os otimizadores de consulta para a avaliação seqüencial da consulta. Primeiro, os modelos de custo são mais complicados, pois os custos de particionamento precisam ser considerados, e questões como distorção e disputa por recursos precisam ser consideradas. Mais importante é a questão de como colocar uma consulta em paralelo. Suponha que tenhamos de alguma forma escolhido uma expressão (entre aquelas equivalentes à consulta) para ser usada na avaliação da consulta. A expressão pode ser representada por uma árvore de operadores, como na primeira seção do Capítulo 13.

Para avaliar uma árvore de operadores em um sistema paralelo, temos de tomar as seguintes decisões:

- Como colocar cada operação em paralelo e quantos processadores usar para isso
- Que operações canalizar por diferentes processadores, que operações executar independentemente em paralelo e que operações executar seqüencialmente, uma após a outra

Essas decisões constituem a tarefa de **escalonamento** da árvore de execução.

Determinar os recursos de cada tipo – como processadores, discos e memória – que deveriam ser alocados a cada operação na árvore é outro aspecto do problema de otimização. Por exemplo, pode parecer sensato usar a quantidade máxima de paralelismo disponível, mas é uma boa idéia não executar certas operações em paralelo. Operações cujos requisitos computacionais são significativamente menores do que o overhead de comunicação devem ser agrupadas com um de seus vizinhos. Caso contrário, a vantagem do paralelismo é anulada pela sobrecarga da comunicação.

Um problema é que canalizações longas não servem para a boa utilização de recursos. A menos que as operações tenham granularidade grossa, a operação final da canalização pode esperar por um longo tempo para receber entradas, enquanto mantém recursos precisos, como memória. Logo, canalizações longas devem ser evitadas.

O número de planos de avaliação paralelos para escolher é muito maior do que o número de planos de avaliação seqüenciais. Otimizar consultas paralelas considerando todas as alternativas, portanto, é muito mais dispendioso do que otimizar consultas seqüenciais. Logo, normalmente adotamos técnicas heurísticas para reduzir o número de planos de execução paralelos que temos de considerar. Descrevemos duas heurísticas populares aqui.

A primeira heurística é considerar apenas planos de avaliação que colocam em paralelo cada operação por todos os processadores, e que não usam qualquer canalização. Essa técnica é usada nas máquinas da série Teradata DBC. Localizar o melhor plano de execução é como fazer otimização de consulta em um sistema seqüencial. As principais diferenças estão em como o particionamento é realizado e que fórmula de estimativa de custo é usada.

A segunda heurística é escolher o plano de avaliação seqüencial mais eficiente e depois colocar as operações em paralelo nesse plano de avaliação. O banco de dados paralelo Volcano popularizou um modelo de paralelismo chamado modelo **operador de troca**. Esse modelo usa implementações existentes de operações, operando sobre cópias locais dos dados, junto com uma operação de troca que movimenta os dados entre diferentes processadores. Os operadores de troca podem ser introduzidos em um plano de avaliação pra transformá-lo em um plano de avaliação paralelo.

Outra dimensão da otimização é o projeto da organização de armazenamento físico para agilizar as consultas. A organização física ideal difere para diferentes consultas. O administrador de banco de dados precisa escolher uma organização física que parece ser boa para a diversidade esperada de consultas de banco de dados. Assim, a área da otimização de consulta paralela é complexa, e ainda é uma área de intensa pesquisa.

Projeto de sistemas paralelos

Até aqui, este capítulo se concentrou no paralelismo de armazenamento de dados e no processamento da consulta. Como os sistemas de banco de dados paralelos em grande escala são usados principalmente para armazenar grandes volumes de dados, e para processar consultas de apoio à decisão sobre esses dados, esses assuntos são os mais importantes em um sistema de banco de dados paralelo. A carga de dados em paralelo a partir de fontes externas é um requisito importante, se tivermos de tratar grandes volumes de dados recebidos.

Um grande sistema de banco de dados paralelo também precisa enfrentar estas questões de disponibilidade:

- Elasticidade à falha de alguns processadores ou discos
- Reorganização on-line de mudanças de dados e esquema

Consideramos essas questões aqui.

Com uma grande quantidade de processadores e discos, a probabilidade de que pelo menos um processador ou disco apresente defeito é muito maior que em um sistema de único processador com um disco. Um sistema paralelo mal projetado deixará de funcionar se qualquer componente (processador ou disco) falhar. Supondo que a probabilidade de falha de um único processador ou disco seja pequena, a probabilidade de falha do sistema sobe linearmente com o número de processadores e discos. Se um único processador ou disco falhasse uma vez a cada cinco anos, um sistema com 100 processadores teria uma falha a cada 18 dias.

Portanto, os sistemas de banco de dados paralelos em grande escala, como Compaq Himalaya, Teradata e Informix XPS (agora uma divisão da IBM), são projetados para operar mesmo que um processador ou disco falhe. Os dados são replicados por pelo menos dois processadores. Se um processador falhar, os dados que ele armazena ainda podem ser acessados a partir dos outros processadores. O sistema registra os processadores que falharam e distribui o trabalho entre os processadores funcionando. As solicitações para dados armazenados no site que falhou são roteadas automaticamente para os sites de backup que armazenam uma réplica dos dados. Se todos os dados de um processador A forem replicados em um único processador B, B terá de lidar com todas as solicitações para A, além daquelas para si mesmo, e isso resultará em B se tornando um gargalo. Portanto, as réplicas dos dados de um processador são particionadas por vários outros processadores.

Quando estamos lidando com grandes volumes de dados (variando na faixa dos terabytes), operações simples, como a criação de índices, e mudanças no esquema, como a inclusão de uma coluna a uma relação, podem levar muito tempo – talvez horas ou até mesmo dias. Portanto, é inaceitável que o sistema de banco de dados esteja indisponível enquanto tais operações estão em progresso. Muitos sistemas de banco de dados paralelos, como os sistemas Compaq Himalaya, permitem que tais operações sejam realizadas on-line, ou seja, enquanto o sistema está executando outras transações.

Considere, por exemplo, a **construção de índice on-line**. Um sistema que admite esse recurso permite inserções, exclusões e atualizações sobre uma relação mesmo quando um índice está sendo criado sobre a relação. A operação de criação de índice, portanto, não pode bloquear a relação inteira no modo compartilhado, como teria feito de outra forma. Em vez disso, o processo registra atualizações que ocorrem enquanto ele está ativo e incorpora as mudanças a um índice sendo construído.

Resumo

- Os bancos de dados paralelos ganharam aceitação comercial significativa nos últimos 20 anos.
- No paralelismo de E/S, as relações são particionadas entre os discos disponíveis para que possam ser apanhadas mais rapidamente. Três técnicas de particionamento comumente usadas são o particionamento de rodízio, particionamento de hash e particionamento de intervalo.
- A distorção é um problema importante, especialmente com maiores graus de paralelismo. Os vetores de particionamento balanceados, usando histogramas, e o particionamento virtual do processador estão entre as técnicas usadas para reduzir a distorção.
- No paralelismo interconsulta, executamos diferentes consultas simultaneamente para aumentar o throughput.
- O paralelismo intraconsulta tenta reduzir o custo da execução de uma consulta. Existem dois tipos de paralelismo intraconsulta: paralelismo intra-operação e paralelismo interoperação.
- Usamos o paralelismo intra-operação para executar operações relacionais, como classificações e junções, em paralelo. O paralelismo intra-operação é natural para operações relacionais, pois não são orientadas a conjunto.
- Existem duas técnicas básicas para executar em paralelo uma operação binária, como uma junção.
  - No paralelismo particionado, as relações são divididas em várias partes, e as tuplas em r_i são juntadas apenas com as tuplas de s_i. O paralelismo particionado só pode ser usado para junções naturais e equijunções.
  - Na técnica fragmentar-e-replicar, as duas relações são particionadas, e cada partição é replicada. Em fragmentar-e-replicar assimétrico, uma das relações é replicada enquanto a outra é particionada. Diferente do paralelismo particionado, fragmentar-e-replicar e fragmentar-e-replicar assimétrico podem ser usados com qualquer condição de junção.

  As duas técnicas de paralelismo podem atuar em conjunto com qualquer técnica de junção.
- No paralelismo independente, diferentes operações que não dependem uma da outra são executadas em paralelo.
- No paralelismo canalizado, os processadores enviam os resultados de uma operação para outra à medida que esses resultados são calculados, sem esperar que a operação inteira termine.
- A otimização da consulta em bancos de dados paralelos é muito mais complexa do que a otimização da consulta em bancos de dados seqüenciais.

Termos de revisão

- Consultas de apoio à decisão
- Paralelismo de E/S
- Particionamento horizontal
- Técnicas de particionamento
  - Rodízio
  - Particionamento de hash
  - Particionamento de intervalo
- Atributo de particionamento
- Vetor de particionamento
- Consulta pontual
- Consulta de intervalo
- Distorção
  - Distorção de execução
  - Distorção de valor de atributo
  - Distorção de partição
- Tratamento da distorção
  - Vetor de particionamento de intervalo balanceado
  - Histograma
  - Processadores virtuais
- Paralelismo interconsulta
- Coerência de cache
- Paralelismo intraconsulta
  - Paralelismo intra-operação
  - Paralelismo interoperação
- Classificação paralela
  - Classificação por particionamento de intervalo
  - Sort-merge externo paralelo
- Paralelismo de dados
- Junção paralela
  - Junção particionada
  - Junção fragmentar-e-replicar
  - Junção fragmentar-e-replicar assimétrica
  - Junção de hash paralela particionada
  - Junção de loop aninhado paralela
- Seleção paralela
- Eliminação de duplicatas paralela
- Projeção paralela
- Agregação paralela
- Custo da avaliação paralela
- Paralelismo interoperação
  - Paralelismo canalizado
  - Paralelismo independente
  - Otimização da consulta
- Escalonamento
- Modelo operador de troca
- Projeto de sistemas paralelos
- Construção de índice on-line

Exercícios práticos

21.1 Em uma seleção de intervalo sobre um atributo particionado por intervalo, é possível que somente um disco possa precisar ser acessado. Descreva os benefícios e as desvantagens dessa propriedade.

21.2 Que forma de paralelismo (interconsulta, interoperação ou intra-operação) provavelmente será o mais importante para cada uma das tarefas a seguir?

a. Aumentar o throughput de um sistema com consultas muito pequenas

b. Aumentar o throughput de um sistema com poucas consultas grandes, quando o número de discos e processadores é grande

21.3 Com o paralelismo canalizado, normalmente é uma boa idéia realizar várias operações em uma pipeline em um único processador, mesmo quando muitos processadores estão disponíveis.

a. Explique por quê.

b. Os argumentos que você usou na parte a seriam mantidos se a máquina tivesse uma arquitetura de memória compartilhada? Explique por que sim ou por que não.

c. Os argumentos na parte a seriam mantidos com o paralelismo independente? (Ou seja, existem casos em que, mesmo que as operações não sejam canalizadas e existam muitos processadores à disposição, ainda é uma boa idéia realizar várias operações no mesmo processador?)

21.4 Considere o processamento da transação usando a técnica fragmentar-e-replicar com particionamento de intervalo. Como você pode otimizar a avaliação se a condição de junção for da forma $|r.A - s.B| \leq k$, onde k é uma constante pequena? Aqui, $|x|$ indica o valor absoluto de x. Uma junção com tal condição de junção é denominada **junção de banda**.

21.5 Lembre-se de que os histogramas são usados para construir partições de intervalo por carga balanceada.

a. Suponha que você tenha um histograma em que os valores estão entre 1 e 100 e são particionados em 10 intervalos, 1-10, 11-20,..., 91-100, com freqüências 15, 5, 20, 10, 10, 5, 5, 20, 5 e 5, respectivamente. Dê uma função de particionamento de intervalo por carga balanceada para dividir os valores em 5 partições.

b. Escreva um algoritmo para calcular uma partição de intervalo balanceado com p partições, dando um histograma de distribuições de freqüência contendo n intervalos.

21.6 Alguns sistemas de banco de dados armazenam uma cópia extra de cada item em discos conectados a um processador diferente, para evitar a perda de dados se um dos processos falhar.

a. Por que é uma boa idéia particionar as cópias dos itens de dados de um processador por vários processadores?

b. Quais são os benefícios e as desvantagens de usar o armazenamento RAID em vez de armazenar uma cópia extra de cada item de dados?

Exercícios

21.7 Para cada uma das três técnicas de particionamento, a saber, rodízio, particionamento de hash e particionamento de intervalo, dê um exemplo de uma consulta para a qual essa técnica de particionamento ofereceria a resposta mais rápida.

21.8 Que fatores poderiam resultar em distorção quando uma relação for particionada em um de seus atributos por:
a. Particionamento de hash?
b. Particionamento de intervalo?
Em cada caso, o que pode ser feito para reduzir a distorção?

21.9 Dê um exemplo de uma junção que não é uma equijunção simples para a qual o paralelismo particionado pode ser usado. Que atributos devem ser usados para o particionamento?

21.10 Descreva uma boa maneira de colocar em paralelo cada um dos seguintes.
a. A operação de diferença
b. Agregação pela operação **count**
c. Agregação pela operação **count distinct**
d. Agregação pela operação avg
e. Junção externa esquerda, se a condição de junção envolver apenas a igualdade
f. Junção externa esquerda, se a condição de junção envolver comparações que não sejam igualdade
g. Junção externa completa, se a condição de junção envolver comparações diferentes da igualdade

21.11 Descreva os benefícios e as desvantagens do paralelismo canalizado.

21.12 Suponha que você queira lidar com uma carga de trabalho consistindo em uma grande quantidade de pequenas transações usando o paralelismo do tipo nada compartilhado.
a. O paralelismo intraconsulta é necessário em tal situação? Se não, por que e que forma de paralelismo é apropriada?
b. Que forma de distorção teria significado com tal carga de trabalho?
c. Suponha que a maioria das transações acessasse um registro de *conta*, que inclui um atributo de tipo de conta, e um registro $mestre_tipo_conta$ associado, que oferece informações sobre o tipo de conta. Como você particionaria e/ou replicaria os dados para agilizar as transações? Você pode supor que a relação $mestre_tipo_conta$ raramente é atualizada.

Notas bibliográficas

Os sistemas de banco de dados relacionais começaram a aparecer no mercado em 1983; agora, eles o dominam. Ao final da década de 1970 e no início da década de 1980, quando o modelo relacional chegou a um patamar razoavelmente seguro, as pessoas reconheceram que os operadores relacionais são altamente paralelizáveis e possuem boas propriedades de fluxo de dados. Um sistema comercial, Teradata, e vários projetos de pesquisa, como GRACE (Kitsuregawa *et al.* [19831, Fushimi *et al.* [19861], GAMMA (DeWitt *et al.* [1986], DeWitt [1990]) e Bubba (Boral *et al.* [1990]), foram iniciados em rápida sucessão. Os pesquisadores usaram esses sistemas de banco de dados paralelos para investigar a praticidade da execução paralela dos operadores relacionais. Subsequentemente, no final da década de 1980 e durante os anos 90, várias outras empresas – como Tandem, Oracle, Sybase, Informix e Red-Brick (agora parte da Informix, que agora faz parte da IBM) – entraram no mercado de banco de dados paralelo. Os projetos de pesquisa no mundo acadêmico incluem XPRS (Stonebraker *et al.* [1989]) e Volcano (Graefe [1990]).

O bloqueio nos bancos de dados paralelos é discutido em Joshi [1991], Mohan e Narang [1991] e Mohan e Narang [1992]. Os protocolos de coerência de cache para sistemas de banco de dados paralelos são discutidos por Dias *et al.* [1989], Mohan e Narang [1991], Mohan e Narang [1992] e Rahm [1993]. Carey *et al.* [1991] discutem as questões de caching em um sistema cliente-servidor. O paralelismo e a recuperação nos sistemas de banco de dados são discutidos por Bayer *et al.* [1980].

Graefe [1993] apresenta um excelente estudo do processamento da consulta, incluindo o processamento paralelo das consultas. O modelo de operador de troca foi defendido por Graefe [1990] e Graefe [1993].

A classificação paralela é discutida em DeWitt *et al.* [1992]. Os algoritmos de junção paralela são descritos por Nakayama *et al.* [1984], Kitsuregawa *et al.* [1983], Richardson *et al.* [1987], Schneider e DeWitt [1989], Kitsuregawa e Ogawa [1990], Lin *et al.* [1994] e Wilschut e outros [1995], entre outros trabalhos. Os algoritmos de junção paralela para arquiteturas de memória compartilhada são descritos por Tsukuda *et al.* [1992], Deshpande e Larson [1992] e Shatdal e Naughton [1993].

O tratamento da distorção em junções paralelas é descrito por Walton *et al.* [1991], Wolf [1991] e DeWitt *et al.* [1992].

As técnicas de otimização de consulta em paralelo são descritas por H. Lu e Tan [1991], Hong e Stonebraker [1991], Ganguly *et al.* [1992], Lanzelotte *et al.* [1993] e Jhingran *et al.* [1997].

CAPÍTULO 22

Bancos de dados distribuídos

Ao contrário dos sistemas paralelos, em que os processadores são bastante acoplados e constituem um único sistema de banco de dados, um sistema de banco de dados distribuído consiste em sites pouco acoplados, que não compartilham componentes físicos. Além do mais, os sistemas de banco de dados que executam em cada site podem ter um grau de independência mútua substancial. Discutimos a estrutura básica dos sistemas distribuídos no Capítulo 20.

Cada site pode participar na execução de transações que acessam dados em um ou vários sites. A diferença principal entre sistemas de banco de dados centralizados e distribuídos é que, no primeiro, os dados residem em um único local, enquanto no segundo, os dados residem em vários locais. Essa distribuição de dados é a causa de muitas dificuldades no processamento de transação e processamento de consulta. Neste capítulo, consideramos essas dificuldades.

Começamos classificando os bancos de dados distribuídos como homogêneos ou heterogêneos, na próxima seção. Depois, consideramos a questão de como armazenar dados em um banco de dados distribuído na seção "Armazenamento de dados distribuído". Na seção "Transações distribuídas", esboçamos um modelo para processamento de transação em um banco de dados distribuído. Na seção "Protocolos commit", descrevemos como implementar transações indivisíveis em um banco de dados distribuído usando protocolos commit especiais. Na seção "Controle de concorrência em bancos de dados distribuídos", descrevemos o controle de concorrência nos bancos de dados distribuídos. Na seção "Disponibilidade", esboçamos como oferecer alta disponibilidade em um banco de dados distribuído explorando a replicação, de modo que o sistema possa continuar processando transações mesmo quando existe uma falha. Consideramos o processamento da consulta nos bancos de dados distribuídos na seção "Processamento de consulta distribuído". Na seção "Bancos de dados distribuídos heterogêneos", esboçamos questões de tratamento de bancos de dados heterogêneos. Na seção "Sistemas de diretório", descrevemos os sistemas de diretório, que podem ser vistos como uma forma especializada de bancos de dados distribuídos.

Bancos de dados homogêneos e heterogêneos

Em um sistema de **banco de dados distribuído homogêneo**, todos os sites possuem software de sistema de gerenciamento de banco de dados idêntico, conhecem um ao outro e concordam em cooperar nas solicitações dos usuários do processamento. Nesse tipo de sistema, os sites locais entregam uma parte de sua autonomia em termos do seu direito de mudar esquemas ou software de sistema de gerenciamento de banco de dados. Esse software também precisa cooperar com outros sites na troca de informações sobre transações, para tornar o processamento da transação possível entre vários sites.

Ao contrário, em um **banco de dados distribuído heterogêneo**, diferentes sites podem usar diferentes esquemas e softwares de sistema de gerenciamento de banco de dados. Os sites podem não ser cientes um do outro, e podem oferecer facilidades apenas limitadas para cooperação no processamento da transação. As diferenças nos esquemas normalmente são um problema importante para o processamento da consulta, enquanto a divergência no software se torna um obstáculo para o processamento de transações que acessam múltiplos sites.

Neste capítulo, vamos nos concentrar em bancos de dados distribuídos homogêneos. Porém, na seção "Bancos de

dados distribuídos heterogêneos", discutimos rapidamente as questões de processamento de consulta em sistemas de banco de dados distribuídos heterogêneos. As questões de processamento de transação em tais sistemas são abordadas mais adiante, na seção "Gerenciamento de transações em bancos de dados múltiplos" do Capítulo 25.

Armazenamento de dados distribuído

Considere uma relação r que deve ser armazenada no banco de dados. Existem duas técnicas para armazenar essa relação no banco de dados distribuído:

- **Replicação.** O sistema mantém várias réplicas (cópias) idênticas da relação e armazena cada uma em um site diferente. A alternativa para a replicação é armazenar apenas uma cópia da relação r.
- **Fragmentação.** O sistema particiona a relação em vários fragmentos e armazena cada fragmento em um site diferente.

Fragmentação e replicação podem ser combinadas: uma relação pode ser particionada em diversos fragmentos e pode haver várias réplicas de cada fragmento. Nas subseções seguintes, elaboramos cada uma dessas técnicas.

Replicação de dados

Se a relação r for replicada, uma cópia da relação r será armazenada em dois ou mais sites. No caso mais extremo, temos replicação completa, em que uma cópia é armazenada em cada site no sistema.

Existem diversas vantagens e desvantagens na replicação.

- **Disponibilidade.** Se um dos sites contendo a relação r falhar, então a relação r pode ser encontrada em outro site. Assim, o sistema pode continuar a processar consultas envolvendo r, apesar da falha de um site.
- **Paralelismo aumentado.** No caso em que a maioria dos acessos à relação r resulta apenas na leitura da relação, então vários sites podem processar consultas envolvendo r em paralelo. Quanto mais réplicas de r houver, maior a chance de que os dados necessários sejam encontrados no site em que a transação está executando. Logo, a replicação de dados diminui o movimento de dados entre os sites.
- **Maior sobrecarga na atualização.** O sistema precisa garantir que todas as réplicas de uma relação r sejam consistentes; caso contrário, computações errôneas podem acontecer. Assim, sempre que r é atualizada, a atualização precisa ser propagada para todos os sites contendo réplicas. O resultado é maior sobrecarga. Por exemplo, em um sistema bancário, em que a informação de conta é replicada em vários sites, é preciso garantir que o saldo em uma conta em particular combine em todos os sites.

Em geral, a replicação garante o desempenho das operações read e aumenta a disponibilidade dos dados às transações somente de leitura. Porém, as transações de atualização incorrem em maior sobrecarga. Controlar as atualizações concorrentes por várias transações aos dados replicados é mais complexo do que em sistemas centralizados, que vimos no Capítulo 16. Podemos simplificar o gerenciamento de réplicas da relação r escolhendo uma delas como a **cópia primária** de r. Por exemplo, em um sistema bancário, uma conta pode ser associada ao site em que foi aberta. De modo semelhante, em um sistema de reserva aérea, um vôo pode estar associado ao site que o origina. Examinaremos o esquema de cópia primária e outras opções para controle de concorrência distribuído na seção "Controle de concorrência em bancos de dados distribuídos".

Fragmentação de dados

Se a relação r for fragmentada, r será dividida em uma série de fragmentos $r_1, r_2, \ldots, r_n$. Esses fragmentos contêm informações suficientes para permitir a reconstrução da relação original r. Existem dois esquemas diferentes para fragmentar uma relação: fragmentação *horizontal* e fragmentação *vertical*. A fragmentação horizontal divide a relação atribuindo cada tupla de r a um ou mais fragmentos. A fragmentação vertical divide a relação decompondo o esquema R da relação r.

Ilustraremos essas técnicas fragmentando a relação *conta*, com o esquema

$$Esquema\_conta = (número\_conta, nome\_agência, saldo)$$

Na **fragmentação horizontal**, uma relação r é particionada em uma série de subconjuntos, $r_1, r_2, \ldots, r_n$. Cada tupla da relação r precisa pertencer a pelo menos um dos fragmentos, de modo que a relação original pode ser reconstruída, se for necessário.

Como ilustração, a relação *conta* pode ser dividida em vários fragmentos diferentes, cada um consistindo em tuplas de contas pertencentes a determinada agência. Se o sistema bancário tiver apenas duas agências – Hillside e Valleyview –, então existem dois fragmentos diferentes:

$$conta_1 = \sigma_{nome\_agência = \text{"Hillside"}}(conta)$$
$$conta_2 = \sigma_{nome\_agência = \text{"Valleyview"}}(conta)$$

A fragmentação horizontal normalmente é usada para manter tuplas nos sites em que são mais usados, para minimizar a transferência de dados.

Em geral, um fragmento horizontal pode ser definido como uma *seleção* sobre a relação global r. Ou seja, usamos um predicado P_i para construir o fragmento r_i:

$$r_i = \sigma_{P_i}(r)$$

Reconstruímos a relação r apanhando a união de todos os fragmentos; ou seja,

$$r = r_1 \cup r_2 \cup \ldots \cup r_n$$

Em nosso exemplo, os fragmentos são disjuntos. Alterando os predicados de seleção usados para construir os fragmentos, podemos ter uma tupla qualquer de r aparecendo em mais de uma de r_i.

Em sua forma mais simples, a fragmentação vertical é igual à decomposição (ver Capítulo 7). A **fragmentação vertical** de $r(R)$ envolve a definição de vários subconjuntos de atributos $R_1, R_2, \ldots, R_n$ do esquema R de modo que

$$R = R_1 \cup R_2 \cup \ldots \cup R_n$$

Cada fragmento r_i de r é definido por

$$r_i = \prod_{R_i}(r)$$

A fragmentação deve ser feita de modo que possamos reconstruir a relação r a partir dos fragmentos apanhando a junção natural

$$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \ldots \bowtie r_n$$

Uma maneira de garantir que a relação r possa ser reconstruída é incluir os atributos de chave primária de R em cada R_i. Geralmente, qualquer superchave pode ser usada. Normalmente, é conveniente acrescentar um atributo especial, chamado *tuple-id*, ao esquema R. O valor de tuple-id de uma tupla é um valor exclusivo que distingue a tupla de todas as outras. O atributo tuple-id, assim, serve como uma chave candidata para o esquema aumentado, e está incluído em cada R_i. O endereço físico ou lógico para uma tupla pode ser usado como um tuple-id, pois cada tupla possui um endereço exclusivo.

Para ilustrar a fragmentação vertical, considere um banco de dados de universidade com uma relação *info_funcionário* que armazena, para cada funcionário, *id_funcionário*, *nome*, *cargo* e *salário*. Por motivos de privacidade, essa relação pode ser fragmentada em uma relação *info_privada_func* contendo atributos *id_funcionário*, *nome* e *cargo*. Estes podem ser armazenados em diferentes sites, novamente por motivos de segurança.

Os dois tipos de fragmentação podem ser aplicados a um único esquema; por exemplo, os fragmentos obtidos pela fragmentação horizontal de uma relação ainda podem ser particionados verticalmente. Os fragmentos também podem ser replicados. Em geral, um fragmento pode ser replicado, as réplicas dos fragmentos podem ser fragmentadas ainda mais, e assim por diante.

Transparência

O usuário de um sistema de banco de dados distribuído não precisa saber onde os dados estão localizados fisicamente ou como eles podem ser acessados no site local específico. Essa característica, chamada **transparência de dados**, pode assumir diversas formas:

- **Transparência da fragmentação.** Os usuários não precisam saber como uma relação foi fragmentada.
- **Transparência da replicação.** Os usuários vêem cada objeto de dados como logicamente exclusivos. O sistema distribuído pode replicar um objeto para aumentar o desempenho do sistema ou a disponibilidade dos dados. Os usuários não precisam se preocupar com quais objetos de dados foram replicados, ou onde as réplicas foram colocadas.
- **Transparência de local.** Os usuários não precisam saber o local físico dos dados. O sistema de banco de dados distribuído deve ser capaz de encontrar quaisquer dados, desde que o identificador de dados seja fornecido pela transação do usuário.

Os itens de dados – como relações, fragmentos e réplicas – precisam ter nomes exclusivos. Essa propriedade é fácil de garantir em um banco de dados centralizado. Porém, em um banco de dados distribuído, temos de ter o cuidado de garantir que dois sites não usem o mesmo nome para itens de dados distintos.

Uma solução para esse problema é exigir que todos os nomes sejam registrados em um **servidor de nomes** central. O servidor de nomes ajuda a garantir que o mesmo nome não seja usado para diferentes itens de dados. Também podemos usá-lo para localizar um item de dados, dado o nome do item. Essa técnica, porém, sofre duas desvantagens importantes. Primeiro, o servidor de nomes pode se tornar um gargalo ao desempenho quando os itens de dados estiverem localizados por seus nomes, resultando em um desempenho fraco. Segundo, se o servidor de nomes falhar, pode não ser possível que qualquer site no sistema distribuído continue a executar.

Uma técnica alternativa mais usada requer que cada site insira como prefixo seu próprio identificador de site a qualquer nome gerado. Essa técnica garante que dois sites não gerarão o mesmo nome (pois cada site tem um identifica-

dor exclusivo). Além do mais, nenhum controle central é exigido. Porém, essa solução não consegue transparência de local, pois os identificadores de site são anexados aos nomes. Assim, a relação *conta* poderia ser referenciada como *site17.conta*, ou *conta@site17*, em vez de simplesmente *conta*. Muitos sistemas de banco de dados utilizam o endereço da Internet de um site para identificá-lo.

Para contornar esse problema, o sistema de banco de dados pode criar um conjunto de nomes alternativos, ou **aliases**, para os itens de dados. Um usuário pode, assim, referenciar os itens de dados por nomes simples, que são traduzidos pelo sistema para nomes completos. O mapeamento de aliases aos nomes reais pode ser armazenado em cada site. Com os aliases, o usuário não precisa saber o local físico de um item de dados. Além do mais, o usuário não será afetado se o administrador de banco de dados decidir mover um item de dados de um site para outro.

Os usuários não precisarão se referir a uma réplica específica de um item de dados. Em vez disso, o sistema deverá determinar quais réplicas referenciar em uma solicitação read e deverá atualizar todas as réplicas em uma solicitação write. Podemos garantir que ele fará isso mantendo uma tabela de catálogo, que o sistema utiliza para determinar todas as réplicas para o item de dados.

Transações distribuídas

O acesso aos diversos itens de dados em um sistema distribuído normalmente é realizado por meio de transações, que precisam preservar as propriedades ACID (primeira seção "" do Capítulo 15). Existem dois tipos de transação que precisamos considerar. As **transações locais** são aquelas que acessam e atualizam dados apenas em um banco de dados local; as **transações globais** são aquelas que acessam e atualizam dados em diversos bancos de dados locais. Garantir as propriedades ACID das transações locais pode ser feito conforme descrevemos nos Capítulos 15, 16 e 17. Porém, para transações globais, essa tarefa é muito mais complicada, pois vários sites podem estar participando na execução. A falha de um desses sites, ou a falha de um enlace de comunicação conectando esses sites, pode resultar em computações errôneas.

Nesta seção, estudamos a estrutura do sistema de um banco de dados distribuído e seus possíveis modos de falha. Com base no modelo apresentado nesta seção, na seção "Protocolos commit" estudamos protocolos para garantir o commit atômico de transações globais, e na seção "Controle de concorrência em bancos de dados distribuídos" estudamos os protocolos para o controle de concorrência nos bancos de dados distribuídos. Na seção "Disponibilidade", estudamos como um banco de dados distribuído pode continuar funcionando mesmo na presença de diversos tipos de falha.

Estrutura do sistema

Cada site tem seu próprio gerenciador de transação *local*, cuja função é garantir as propriedades ACID daquelas transações que executam nesse site. Os diversos gerenciadores de transação cooperam para executar as transações globais. Para entender como esse gerenciador pode ser implementado, considere um modelo abstrato de um sistema de transação, em que cada site contém dois subsistemas:

- O **gerenciador de transações** controla a execução daquelas transações (ou subtransações) que acessam dados armazenados em um site local. Observe que cada transação desse tipo pode ser uma transação local (ou

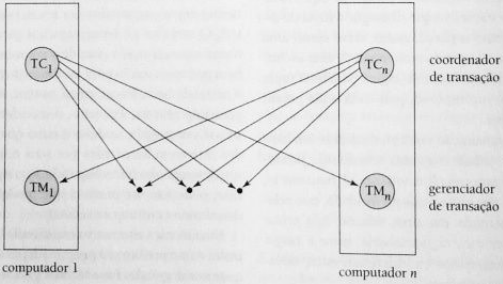


Figura 22.1 Arquitetura do sistema.

seja, uma transação que executa apenas nesse site) ou parte de uma transação global (ou seja, uma transação que é executada em diversos sites).
- O **coordenador de transação** coordena a execução de diversas transações (locais e globais) iniciadas nesse site.

A arquitetura geral do sistema aparece na Figura 22.1.

A estrutura de um gerenciador de transação é semelhante em muitos aspectos à estrutura de um sistema centralizado. Cada gerenciador de transação é responsável por

- Manutenção de um log para fins de recuperação
- Participação em um esquema de controle de concorrência para coordenar a execução concorrente das transações executando nesse site

Como veremos, precisamos modificar os esquemas de recuperação e concorrência para acomodar a distribuição de transações.

O subsistema coordenador de transação não é necessário no ambiente centralizado, pois uma transação acessa dados apenas em um único site. Um coordenador de transação, como seu nome indica, é responsável por coordenar a execução de todas as transações iniciadas nesse site. Para cada transação desse tipo, o coordenador é responsável por

- Iniciar a execução da transação
- Dividir a transação em uma série de subtransações e distribuí-las aos sites apropriados para execução
- Coordenar o término da transação, que pode resultar na transação sendo confirmada em todos os sites ou abortada em todos os sites

Modos de falha do sistema

Um sistema distribuído pode sofrer com os mesmos tipos de falha que um sistema centralizado (por exemplo, erros de software, erros de hardware ou falhas de disco). Porém, existem outros tipos de falha com que precisamos lidar em um ambiente distribuído. Os tipos de falha básicos são

- Falha de um site
- Perda de mensagens
- Falha de um enlace de comunicação
- Partição de rede

A perda ou a adulteração de mensagens sempre é uma possibilidade em um sistema distribuído. O sistema usa os protocolos de controle de transmissão, como TCP/IP, para lidar com esses erros. As informações sobre tais protocolos poderão ser encontradas nos livros-texto padrão sobre redes (consulte as notas bibliográficas).

Porém, se dois sites A e B não estiverem conectados diretamente, as mensagens de um para o outro precisam ser *roteadas* por meio de uma sequência de enlaces de comunicação. Se um enlace de comunicação falhar, as mensagens que teriam sido transmitidas pelo enlace precisam ser roteadas novamente. Em alguns casos, é possível encontrar outra rota pela rede, de modo que as mensagens são capazes de alcançar seu destino. Em outros casos, uma falha pode resultar em não haver conexão entre alguns pares de sites. Um sistema é **particionado** se tiver sido dividido em dois (ou mais) subsistemas, chamados **partições**, que não possuem qualquer conexão entre eles. Observe que, sob essa definição, um subsistema pode consistir em um único nó.

Protocolos commit

Se tivermos de garantir a atomicidade, todos os sites em que a transação T executou precisam combinar com o resultado final da execução. T precisa confirmar em todos os sites, ou precisa abortar em todos os sites. Para garantir essa propriedade, o coordenador da transação de T precisa executar um *protocolo commit*.

Entre os protocolos commit mais simples e mais usados está o **protocolo commit de duas fases** (2PC), descrito na próxima seção. Uma alternativa é o **protocolo commit de três fases** (3PC), que evita certas desvantagens do protocolo 2PC, mas aumenta a complexidade e a sobrecarga. A seção "Commit de três fases" esboça rapidamente o protocolo 3PC.

Commit de duas fases

Primeiro, descrevemos como o protocolo commit de duas fases (2PC) opera durante a operação normal, depois descrevemos como ele trata de falhas e finalmente como ele executa o controle de recuperação e concorrência.

Considere uma transação T iniciada no site S_i, em que o coordenador da transação é C_i.

O protocolo commit

Quando T termina sua execução – ou seja, quando todos os sites em que T executou informam a C_i que T completou – C_i inicia o protocolo 2PC.

- **Fase 1.** C_i acrescenta o registro <prepare T> ao log e o força o log para o armazenamento estável. Depois, ele envia uma mensagem `prepare` T a todos os sites em que T foi executado. Ao receber tal mensagem, o gerenciador de transação nesse site determina se deseja confirmar sua parte de T. Se a resposta for não, ele acrescenta um registro <no T> ao log e depois respon-

de enviando uma mensagem `abort` T para C_i. Se a resposta for sim, ele acrescenta um registro <`ready` T> ao log e força o log (com todos os registros de log correspondentes a T) para o armazenamento estável. O gerenciador de transação, então, responde com uma mensagem `ready` T para C_i.

- **Fase 2.** Quando C_i recebe respostas à mensagem `prepare` T de todos os sites, ou quando um intervalo de tempo pré-especificado tiver passado desde que a mensagem `prepare` T foi enviada, C_i pode determinar se a transação T pode ser confirmada ou abortada. A transação T pode ser confirmada se C_i recebeu uma mensagem `ready` T de todos os sites participantes. Caso contrário, a transação T precisa ser abortada. Dependendo do veredicto, um registro <`commit` T> ou um registro <`abort` T> é acrescentado ao log e o log é forçado para o armazenamento estável. Nesse ponto, o destino da transação foi selado. Depois desse ponto, o coordenador envia uma mensagem commit T ou abort T para todos os sites participantes. Quando um site recebe essa mensagem, ele registra a mensagem no log.

Um site em que T foi executado pode abortar T incondicionalmente a qualquer momento antes de enviar a mensagem `ready` T ao coordenador. Quando a mensagem for enviada, a transação é considerada no **estado pronto** no site. A mensagem `ready` T, com efeito, é uma promessa feita por um site de seguir a ordem do coordenador para confirmar T ou abortar T. Para fazer essa promessa, a informação necessária precisa ser armazenada no armazenamento estável. Caso contrário, se o site falhar depois de enviar `ready` T, ele pode ser incapaz de cumprir sua promessa. Além do mais, os bloqueios adquiridos pela transação precisam continuar sendo mantidos até que a transação termine.

Como a unanimidade é exigida para a confirmação de uma transação, o destino de T está selado assim que pelo menos um site responde `abort` T. Como o site coordenador S_i é um dos sites em que T foi executada, o coordenador pode decidir abortar T unilateralmente. O veredicto final com relação a T é determinado no momento em que o coordenador escreve esse veredicto (confirmar ou abortar) no log e o força para o armazenamento estável. Em algumas implementações do protocolo 2PC, um site envia uma mensagem `acknowledge` T ao coordenador no final da segunda fase do protocolo. Quando o coordenador recebe a mensagem acknowledge T de todos os sites, ele acrescenta o registro <`complete` T> ao log.

Tratamento de falhas

O protocolo 2PC responde de diferentes maneiras a vários tipos de falhas:

- **Falha de um site participante.** Se o coordenador C_i detectar que um site falhou, ele toma estas ações: se o site falhar antes de responder com uma mensagem `ready` T para C_i, o coordenador considera que ele respondeu com uma mensagem `abort` T. Se o site falhar depois que o coordenador tiver recebido a mensagem `ready` T do site, o coordenador executa o restante do protocolo commit pela forma normal, ignorando a falha do site.

  Quando um site participante S_k se recupera de uma falha, ele precisa examinar seu log para determinar o destino das transações que estavam no meio da execução quando a falha ocorreu. Considere que T seja uma transação desse tipo. Consideramos cada um dos casos possíveis:
  - O log contém um registro <`commit` T>. Nesse caso, o site executa `redo`(T).
  - O log contém um registro <`abort` T>. Nesse caso, o site executa `undo`(T).
  - O log contém um registro <`ready` T>. Nesse caso, o site precisa consultar C_i para determinar o destino de T. Se C_i estiver ativo, ele notifica S_k com relação a se T confirmou ou abortou. No primeiro caso, ele executa `redo`(T); senão, ele executa `undo`(T). Se C_i estiver parado, S_k precisa tentar encontrar o destino de T a partir de outros sites. Ele faz isso enviando uma mensagem `querystatus` T a todos os sites no sistema. No recebimento de uma mensagem, um site precisa consultar seu log para determinar se T foi executado lá, e se tiver sido, se T confirmou ou abortou. Depois, ele notifica S_k sobre seu resultado. Se nenhum site tiver a informação apropriada (ou seja, se T confirmou ou abortou), então S_k pode nem abortar nem confirmar T. A decisão referente a T é adiada até que S_k possa obter a informação necessária. Assim, S_k precisa reenviar periodicamente a mensagem `querystatus` aos outros sites. Ele continua a fazer isso até que um site que contém a informação necessária se recupere. Observe que o site em que C_i reside sempre tem a informação necessária.
  - O log não contém registros de controle (`abort`, `commit`, `ready`) referentes a T. Assim, sabemos que S_k falhou antes de responder à mensagem `prepare` T de C_i. Como a falha de S_k impede o envio de tal resposta, por nosso algoritmo, C_i precisa abortar T. Logo, S_k precisa executar `undo`(T).
- **Falha do coordenador.** Se o coordenador falhar no meio da execução do protocolo commit para a transação T, então os sites participantes precisam decidir o destino de T. Veremos que, em certos casos, os sites participantes não podem decidir se confirmarão ou abortarão T, e portanto esses sites precisam esperar pela recuperação do coordenador que falhou.

- Se um site ativo tiver um registro <commit T>, então T precisa ser confirmada.
- Se um site ativo tiver um registro <abort T> em seu log, então T precisa ser abortada.
- Se algum site ativo não tiver um registro <ready T> em seu log, então o coordenador que falhou C_i não pode ter decidido confirmar T, pois um site que não tem um registro <ready T> em seu log não pode ter enviado uma mensagem ready T para C_i. Porém, o coordenador pode ter decidido abortar T, mas não confirmar T. Em vez de esperar que C_i se recupere, é preferível abortar T.
- Se nenhum dos casos anteriores for verdadeiro, então todos os sites ativos precisam ter um registro <ready T> em seus logs, mas nenhum registro de controle adicional (como <abort T> ou <commit T>). Como o coordenador falhou, é impossível determinar se uma decisão foi tomada, e se tiver sido, qual é a decisão, até que o coordenador se recupere. Assim, os sites ativos precisam esperar que C_i se recupere. Como o destino de T permanece em dúvida, T pode continuar a manter recursos do sistema. Por exemplo, se o bloqueio for usado, T pode manter bloqueios sobre os dados nos sites ativos. Tal situação é indesejável, pois podem se passar horas ou dias antes que C_i esteja novamente ativo. Durante esse tempo, outras transações podem ser forçadas a esperar por T. Como resultado, os itens de dados podem estar indisponíveis não apenas no site que falhou (C_i), mas também nos sites ativos. Essa situação é chamada de problema de **bloqueio**, pois T está bloqueada dependendo da recuperação de C_i.

- **Partição de rede.** Quando uma rede se divide, existem duas possibilidades.

1. O coordenador e todos os seus participantes permanecem em uma partição. Nesse caso, a falha não tem efeito sobre o protocolo commit.
2. O coordenador e seus participantes pertencem a várias partições. Do ponto de vista dos sites em uma das partições, parece que os sites em outras partições falharam. Os sites que não estão na partição contendo o coordenador simplesmente executam o protocolo para lidar com a falha do coordenador. O coordenador e os sites que estão na mesma partição do coordenador seguem o protocolo de commit normal, supondo que os sites em outras partições falharam.

Assim, a principal desvantagem do protocolo 2PC é que a falha do coordenador pode resultar em bloqueio, em que uma decisão de confirmar ou abortar T pode ter de ser adiada até que C_i se recupere.

Controle de recuperação e concorrência

Quando um site que falhou é reiniciado, podemos realizar a recuperação usando, por exemplo, o algoritmo de recuperação descrito na seção "Técnicas de recuperação avançadas" do Capítulo 17. Para lidar com os protocolos commit distribuídos (como 2PC e 3PC), o procedimento de recuperação precisa tratar **transações em dúvida** de forma especial; transações em dúvida são transações para as quais um registro de log <ready T> é encontrado, mas nem um registro de log <commit T> nem um registro de log <abort T> é encontrado. O site em recuperação precisa determinar o status commit-abort dessas transações conectando outros sites, conforme descrevemos na seção "Tratamento de falhas".

Porém, se a recuperação for feita como descrevemos, o processamento de transação normal no site não pode começar até que todas as transações em dúvida tenham sido confirmadas ou revertidas. A descoberta do status de transações em dúvida pode ser lenta, pois vários sites podem ter de ser contatados. Além do mais, se o coordenador tiver falhado e nenhum outro site tiver informações sobre o status commit-abort de uma transação incompleta, a recuperação potencialmente poderia se tornar bloqueada se o 2PC for usado. Como resultado, o site realizando a recuperação na partida pode permanecer inutilizável por um longo período.

Para contornar esse problema, os algoritmos de recuperação normalmente oferecem suporte para observar informações de bloqueio no log. (Estamos supondo aqui que o bloqueio é usado para controle de concorrência.) Em vez de escrever um registro de log <ready T>, o algoritmo escreve um registro de log <ready T, L>, onde L é uma lista de todos os bloqueios de leitura mantidos pela transação T quando o registro de log for escrito. No momento da recuperação, depois de realizar as ações de recuperação locais, para cada transação em dúvida T, todos os bloqueios de escrita observados no registro de log <ready T, L> (lido do log) são readquiridos.

Depois que a reaquisição de bloqueio estiver completa para todas as transações em dúvida, o processamento de transações pode iniciar no site, mesmo antes que o status commit-abort das transações em dúvida seja determinado. O commit ou rollback de transações em dúvida prossegue de forma simultânea com a execução de novas transações. Assim, a recuperação do site é mais rápida, e nunca é bloqueada. Observe que novas transações que possuem um conflito de bloqueio com quaisquer bloqueios de escrita mantidos por transações em dúvida serão incapazes de fazer progresso até que as transações em dúvida conflitantes tenham sido confirmadas ou revertidas.

Commit de três fases

O protocolo commit de três fases (3PC) é uma extensão do protocolo commit de duas fases, que evita o problema de

bloqueio sob certas suposições. Em particular, considera-se que não ocorre qualquer partição de rede, e não mais do que k sites falham, onde k é algum número predeterminado. Sob essas suposições, o protocolo evita o bloqueio introduzindo uma terceira fase extra, em que vários sites estão envolvidos na decisão de confirmar. Em vez de observar diretamente a decisão de commit em seu armazenamento persistente, o coordenador primeiro garante que pelo menos k outros sites saibam que ele pretendia confirmar a transação. Se o coordenador falhar, os sites restantes primeiro selecionam um novo coordenador. Esse novo coordenador verifica o status do protocolo a partir dos sites restantes; se o coordenador tiver decidido confirmar, pelo menos um dos outros k sites que ele informou estarão ativos e garantirão que a decisão de commit será respeitada. O novo coordenador reinicia a terceira fase do protocolo se algum site souber que o coordenador antigo desejava confirmar a transação. Caso contrário, o novo coordenador aborta a transação.

Embora o protocolo 3PC tenha a propriedade desejável de não bloquear a menos que k sites falhem, ele tem a desvantagem de que um particionamento da rede parecerá ser o mesmo caso tal que mais que k sites falhem, o que levaria ao bloqueio. O protocolo também precisa ser implementado cuidadosamente para garantir que o particionamento da rede (ou mais que k sites falhando) não resulte em inconsistências, em que uma transação é confirmada em uma partição e abortada em outra. Devido à sua sobrecarga, o protocolo 3PC não é muito usado. Veja as notas bibliográficas e obtenha referências com mais detalhes do protocolo 3PC.

Modelos alternativos de processamento de transação

Para muitas aplicações, o problema de bloqueio do commit de duas fases não é aceitável. O problema aqui é a noção de uma única transação que funciona por vários sites. Nesta seção, descrevemos como usar as mensagens persistentes para evitar o problema de commit distribuído, e depois esboçamos rapidamente a questão maior dos fluxos de trabalho; fluxos de trabalho são considerados com mais detalhes na seção "Fluxos de trabalho transacionais" do Capítulo 25.

Para entender as mensagens persistentes, considere como alguém poderia transferir fundos entre dois bancos diferentes, cada um com seu próprio computador. Uma técnica é ter uma transação englobando dois sites e usar o commit de duas fases para garantir a atomicidade. Porém, a transação pode ter de atualizar o saldo total do banco, e o bloqueio poderia ter um impacto sério sobre todas as outras transações em cada banco, pois quase todas as transações no banco atualizariam o saldo total do banco.

Ao contrário, considere como ocorre a transferência de fundos por um cheque bancário. Primeiro, o banco deduz a quantia do cheque do saldo disponível e emite o cheque. O cheque então é transferido fisicamente para o outro banco, onde é depositado. Depois de verificar o cheque, o banco aumenta o saldo local pela quantia do cheque. O cheque constitui uma mensagem enviada entre os dois bancos. Para que os fundos não sejam perdidos ou aumentados incorretamente, o cheque não pode ser perdido e não pode ser duplicado ou depositado mais de uma vez. Quando os computadores do banco estão conectados por uma rede, as mensagens persistentes oferecem o mesmo serviço que o cheque (mas muito mais rapidamente, é claro).

Mensagens persistentes são mensagens que têm garantia de serem entregues ao destinatário exatamente uma vez (nem menos, nem mais), independente de falhas, se a transação enviando a mensagem for confirmada, e têm garantias de não serem entregues se a transação abortar. Técnicas de recuperação de banco de dados são usadas para implementar as mensagens persistentes em cima dos canais de rede normais, como veremos em breve. Ao contrário, as mensagens normais podem ser perdidas ou até mesmo entregues várias vezes em algumas situações.

O tratamento de erro é mais complicado com as mensagens persistentes do que com o commit de duas fases. Por exemplo, se a conta onde o cheque deve ser depositado tiver sido fechada, o cheque precisa ser enviado de volta à conta de origem e creditado de volta. Os dois sites, portanto, precisam ser fornecidos com o código de tratamento de erro, junto com o código para lidar com as mensagens persistentes. Ao contrário, com o commit de duas fases, o erro seria detectado pela transação, que nunca deduziria o valor em primeiro lugar.

Os tipos de condições de exceção que podem surgir dependem da aplicação, de modo que não é possível que o sistema de banco de dados trate de exceções automaticamente. Os programas de aplicação que enviam e recebem mensagens persistentes precisam incluir código para lidar com condições de exceção e trazer o sistema de volta a um estado consistente. por exemplo, não é aceitável apenas perder o dinheiro sendo transferido se a conta credora tiver sido fechada; o dinheiro precisa ser creditado de volta à conta devedora, e se isso não for possível por algum motivo, alguém terá de ser alertado para resolver a situação manualmente.

Existem muitas aplicações em que o benefício de eliminar o bloqueio compensa o esforço extra para implementar sistemas que usam mensagens persistentes. De fato, poucas organizações concordariam em dar suporte ao commit de duas fases para transações originando fora da organização, pois falhas poderiam acontecer no bloqueio de acesso aos dados locais. As mensagens persistentes, portanto, desempenham um papel importante na execução de transações que cruzam limites organizacionais.

Fluxos de trabalho oferecem um modelo geral de processamento de transação envolvendo vários sites e possivelmente o processamento humano de certas etapas. Por exemplo, quando um banco recebe um pedido de empréstimo, existem muitas etapas necessárias, incluindo o contato com agências externas de verificação de crédito, antes de aprovar ou rejeitar um pedido de empréstimo. As etapas, juntas, formam um fluxo de trabalho. Estudamos os fluxos de trabalho com mais detalhes na seção "Fluxos de trabalho transacionais" do Capítulo 25. Também observamos que as mensagens persistentes formam a base para os fluxos de trabalho em um ambiente distribuído.

Agora, consideramos a **implementação** das mensagens persistentes. As mensagens persistentes podem ser implementadas com base em uma infra-estrutura de mensagem não confiável, que pode perder mensagens ou entregá-las várias vezes, por estes protocolos:

- **Protocolo do site emissor.** Quando uma transação deseja enviar uma mensagem persistente, ela escreve um registro contendo a mensagem em uma relação especial *messages_to_send*, em vez de enviar diretamente a mensagem. A mensagem também recebe um identificador exclusivo de mensagem.

  Um *processo de entrega de mensagem* monitora a relação e, quando uma nova mensagem é encontrada, ele envia a mensagem ao seu destino. Os mecanismos normais de controle de concorrência de banco de dados garantem que o processo do sistema leia a mensagem somente depois que a transação que a escreveu confirmar; se a transação abortar, o mecanismo de recuperação normal exclui a mensagem da relação.

  O processo de remessa de mensagem só exclui uma mensagem da relação depois de receber uma confirmação do site de destino. Se não receber confirmação do site de destino, após algum tempo, ele envia a mensagem novamente, e repete a ação até que uma confirmação seja recebida.

  Em caso de falhas permanentes, o sistema decidirá, depois de algum período de tempo, que a mensagem não pode ser entregue. O código de tratamento de exceção fornecido pela aplicação é então invocado para lidar com a falha. A escrita da mensagem em uma relação e seu processamento apenas depois que a transação confirmar garante que a mensagem será entregue se e somente se ela for confirmada. Enviá-la repetidamente garante que será entregue mesmo que existam (temporariamente) falhas do sistema ou da rede.
- **Protocolo do site receptor.** Quando um site recebe uma mensagem persistente, ele executa uma transação que acrescenta a mensagem a uma relação especial *received_messages*, desde que ainda não esteja presente na relação (o identificador de mensagem exclusivo detecta duplicatas). Depois que a transação for confirmada, ou se a mensagem já estava presente na relação, o site receptor envia uma confirmação de volta ao site emissor.

  Observe que não é seguro enviar a confirmação antes da confirmação da transação, pois uma falha do sistema pode resultar em perda da mensagem. Verificar se a mensagem foi recebida anteriormente é essencial para evitar várias remessas da mensagem.

  Em muitos sistemas de mensagem, é possível que as mensagens sejam adiadas arbitrariamente, embora esses atrasos sejam muito improváveis. Portanto, por segurança, a mensagem nunca deve ser excluída da relação *received_messages*. Sua exclusão poderia resultar em uma remessa duplicada não sendo detectada. Como resultado, a relação *received_messages* pode crescer indefinidamente. Para resolver esse problema, cada mensagem recebe um timestamp, e se o timestamp de uma mensagem recebida for mais antigo que algum tempo limite, a mensagem será descartada. Todas as mensagens registradas na relação *received_messages* que são mais antigas que o tempo limite podem ser excluídas.

Controle de concorrência em bancos de dados distribuídos

Mostramos aqui como alguns dos esquemas de controle de concorrência discutidos no Capítulo 16 podem ser modificados de modo que possam ser usados em um ambiente distribuído. Consideramos que cada site participa na execução de um protocolo commit para garantir a atomicidade de transação global.

Os protocolos que descrevemos nesta seção exigem que as atualizações sejam feitas em todas as réplicas de um item de dados. Se qualquer site contendo uma réplica de um item de dados tiver falhado, as atualizações no item de dados não podem ser processadas. Na seção "Disponibilidade", descrevemos protocolos que podem continuar o processamento da transação mesmo que alguns sites ou enlaces tenham falhado, oferecendo assim uma alta disponibilidade.

Protocolos de bloqueio

Os diversos protocolos de bloqueio descritos no Capítulo 16 podem ser usados em um ambiente distribuído. A única mudança que precisa ser incorporada é o modo como o gerenciador de bloqueio lida com os dados replicados. Apresentamos vários esquemas possíveis que se aplicam a um ambiente em que os dados podem ser replicados em vários sites. Como no Capítulo 16, vamos considerar a existência dos modos de bloqueio *compartilhado* e *exclusivo*.

Técnica de gerenciador de bloqueio único

Na técnica de **gerenciador de bloqueio único**, o sistema mantém um *único* gerenciador de bloqueio que reside em um *único* site escolhido – digamos, S_i. Todas as solicitações de bloqueio e desbloqueio são feitas no site S_i. Quando uma transação precisa bloquear um item de dados, ele envia uma solicitação de bloqueio para S_i. O gerenciador de bloqueio determina se o bloqueio pode ser concedido imediatamente. Se o bloqueio pode ser concedido, o gerenciador de bloqueio envia uma mensagem com esse efeito para o site em que a solicitação de bloqueio foi iniciada. Caso contrário, a solicitação é adiada até que possa ser concedida, quando uma mensagem é enviada ao site em que a solicitação de bloqueio foi iniciada. A transação pode ler o item de dados de *qualquer* um dos sites em que reside uma réplica do item de dados. No caso de uma escrita, todos os sites em que reside uma réplica do item de dados precisam estar envolvidos na escrita.

O esquema tem estas vantagens:

- **Implementação simples.** Esse esquema requer duas mensagens para tratar de solicitações de bloqueio e uma mensagem para tratar de solicitações de desbloqueio.
- **Tratamento de impasse simples.** Como todas as solicitações de bloqueio e desbloqueio são feitas em um site, os algoritmos de tratamento de impasse discutidos no Capítulo 16 podem ser aplicados diretamente a esse ambiente.

As desvantagens do esquema são:

- **Gargalo.** O site S_i se torna um gargalo, pois todas as solicitações precisam ser processadas lá.
- **Vulnerabilidade.** Se o site S_i falhar, o controlador de concorrência está perdido. O processamento precisa parar ou um esquema de recuperação precisa ser usado para que um site de backup possa assumir o gerenciamento de bloqueio a partir de S_i, conforme descrevemos na seção "Seleção do coordenador".

Gerenciador de bloqueio distribuído

Um compromisso entre as vantagens e desvantagens pode ser alcançado por meio da técnica de **gerenciador de bloqueio distribuído**, em que a função do gerenciador de bloqueio é distribuída por vários sites.

Cada site mantém um gerenciador de bloqueio local cuja função é administrar as solicitações de bloqueio e desbloqueio para aqueles itens de dados que estão armazenados nesse site. Quando uma transação deseja bloquear o item de dados Q, que não é replicado e reside no site S_i, uma mensagem é enviada ao gerenciador de bloqueio no site S_i solicitando um bloqueio (em determinado modo de bloqueio). Se o item de dados Q estiver bloqueado em um modo incompatível, então a solicitação é adiada até que possa ser concedida. Quando tiver sido determinado que a solicitação de bloqueio pode ser concedida, o gerenciador de bloqueio envia uma mensagem de volta a quem a iniciou, indicando que concedeu a solicitação de bloqueio.

Existem várias maneiras alternativas de lidar com a replicação de itens de dados, que estudamos nas seções "Cópia primária" a "Protocolo de consenso de quórum".

O esquema do gerenciador de bloqueio distribuído tem a vantagem de implementação simples, e reduz o grau ao qual o coordenador é um gargalo. Ele tem uma sobrecarga razoavelmente baixa, exigindo duas transferências de mensagem para lidar com solicitações de bloqueio e uma transferência de mensagem para tratar de solicitações de desbloqueio. Porém, o tratamento de impasse é mais complexo, pois as solicitações de bloqueio e desbloqueio não são mais feitas em um único site: pode haver impasses entre sites mesmo quando não existe impasse dentro de um único site. Os algoritmos de tratamento de impasse discutidos no Capítulo 16 precisam ser modificados, como discutiremos na seção "Tratamento de impasse", para detectar os impasses globais.

Cópia primária

Quando um sistema usa a replicação de dados, podemos escolher uma das réplicas como **cópia primária**. Assim, para cada item de dados Q, a cópia primária de Q precisa residir exatamente em um site, que chamamos de **site primário** de Q.

Quando uma transação precisa bloquear um item de dados Q, ela solicita um bloqueio no site primário de Q. Como antes, a resposta à solicitação é adiada até que possa ser concedida.

Assim, a cópia primária permite que o controle de concorrência para dados replicados seja tratado como o controle de concorrência para dados não replicados. Essa semelhança leva em conta uma implementação simples. Porém, se o site primário de Q falhar, Q fica inacessível, embora outros sites contendo uma réplica possam ser acessíveis.

Protocolo da maioria

O **protocolo da maioria** funciona desta maneira: se o item de dados Q for replicado em n diferentes sites, então uma mensagem de solicitação de bloqueio precisa ser enviada a mais de metade dos n sites em que Q está armazenado. Cada gerenciador de bloqueio determina se o bloqueio pode ser concedido imediatamente (por parte dele). Como antes, a resposta é adiada até que a solicitação possa ser concedida. A transação não opera sobre Q até que tenha obtido com sucesso um bloqueio sobre a maioria das réplicas de Q.

Consideramos por enquanto que as escritas são realizadas sobre todas as réplicas, exigindo que todos os sites contendo réplicas estejam disponíveis. Porém, o principal benefício do protocolo da maioria é que ele pode ser estendido para lidar com falhas dos sites, como veremos na seção "Técnica baseada na maioria". O protocolo também lida com dados replicados de uma maneira descentralizada, evitando, assim, as desvantagens do controle central. Porém, ele sofre com estas desvantagens:

- **Implementação.** O protocolo da maioria é mais complicado de implementar que os esquemas anteriores. Ele exige pelo menos $2(n/2 + 1)$ mensagens para tratar de solicitações de bloco e pelo menos $(n/2 + 1)$ mensagens para tratar de solicitações de desbloqueio.
- **Tratamento de impasse.** Além do problema de impasses globais devido ao uso de uma técnica de gerenciador de bloqueio distribuído, é possível que um impasse ocorra mesmo que somente um item de dados esteja sendo bloqueado. Como ilustração, considere um sistema com quatro sites e replicação total. Suponha que as transações T_1 e T_2 queiram bloquear o item de dados Q no modo exclusivo. A transação T_1 pode ter sucesso no bloqueio de Q nos sites S_1 e S_3, enquanto a transação T_2 pode ter sucesso no bloqueio de Q nos sites S_2 e S_4. Cada um, então, precisa esperar para adquirir o terceiro bloqueio; logo, um impasse aconteceu. Felizmente, podemos evitar esses impasses com relativa facilidade, exigindo que todos os sites solicitem bloqueios sobre as réplicas de um item de dados na mesma ordem predeterminada.

Protocolo parcial

O protocolo **parcial** é outra técnica para lidar com a replicação. A diferença do protocolo da maioria é que as solicitações para bloqueios compartilhados recebem um tratamento mais favorável do que as solicitações para bloqueios exclusivos.

- **Bloqueios compartilhados.** Quando uma transação precisa bloquear um item de dados Q, ela simplesmente solicita um bloqueio sobre Q a partir do gerenciador de bloqueio em um site que contém uma réplica de Q.
- **Bloqueios exclusivos.** Quando uma transação precisa bloquear o item de dados Q, ela solicita um bloqueio sobre Q a partir do gerenciador de bloqueio em todos os sites que contêm uma réplica de Q.

Como antes, a resposta à solicitação é adiada até que possa ser concedida.

O esquema parcial tem a vantagem de impor menos sobrecarga sobre operações `ready` do que o protocolo da maioria. Essa economia é especialmente significativa em casos comuns em que a frequência de `ready` é muito maior do que a frequência de `write`. Porém, a sobrecarga adicional sobre escritas é uma desvantagem. Além do mais, o protocolo parcial compartilha a desvantagem do protocolo da maioria da complexidade no tratamento de impasse.

Protocolo de consenso de quórum

O protocolo de **consenso de quórum** é uma generalização do protocolo da maioria. O protocolo de consenso comum atribui a cada site um peso não negativo. Ele atribui dois inteiros às operações de leitura e escrita sobre um item x, chamados **quórum de leitura** Q_r e **quórum de escrita** Q_w, que precisam satisfazer a condição a seguir, em que S é o peso total de todos os sites em que x reside:

$$Q_r + Q_w > S \textbf{ and } 2 * Q_w > S$$

Para executar uma operação de leitura, réplicas suficientes precisam ser bloqueadas para que seu peso total seja $\geq Q_r$. Para executar uma operação de escrita, réplicas suficientes precisam ser bloqueadas de modo que seu peso total seja $\geq Q_w$.

Um benefício da técnica de consenso de quórum é que ela pode permitir que o custo do bloqueio de leitura ou escrita seja reduzido seletivamente pela definição apropriada dos quóruns de leitura e escrita. Por exemplo, com um quórum de leitura pequeno, as leituras precisam obter menos bloqueios, mas o quórum de escrita será maior, daí as escritas precisarem obter mais bloqueios. Além disso, se pesos maiores forem dados a alguns sites (por exemplo, aqueles menos prováveis de falhar), menos sites precisarão ser acessados para adquirir bloqueios. De fato, definindo pesos e quóruns corretamente, o protocolo de consenso de quórum pode simular o protocolo da maioria e os protocolos parciais.

Como o protocolo da maioria, o consenso de quórum pode ser estendido para funcionar mesmo na presença de falhas dos sites, como veremos na seção "Técnica baseada na maioria".

Timestamp

A idéia principal por trás do esquema de timestamp na seção "Protocolos baseados em timestamp" do Capítulo 16 é que cada transação receba um timestamp *exclusivo*, que o sistema usa na decisão da ordem de seriação. Nossa primeira tarefa, então, na generalização do esquema centralizado para um esquema distribuído é desenvolver um esquema para gerar timestamps exclusivos. Depois, os diversos protocolos podem operar diretamente ao ambiente não replicado.

Existem dois métodos principais para gerar timestamps exclusivos, um centralizado e um distribuído. No esquema centralizado, um único site distribui os timestamps. O site

pode usar um contador lógico ou seu próprio clock local para essa finalidade.

No esquema distribuído, cada site gera um timestamp local exclusivo usando um contador lógico ou o clock local. Obtemos o timestamp global exclusivo concatenando o timestamp local exclusivo com o identificador do site, que também precisa ser exclusivo (Figura 22.2). A ordem de concatenação é importante! Usamos o identificador do site na posição menos significativa para garantir que os timestamps globais gerados em um site nem sempre sejam maiores do que aqueles gerados em outro site. Compare essa técnica para gerar timestamps exclusivos com aquela que apresentamos na seção "Transparência", para gerar nomes exclusivos.

Ainda podemos ter um problema se um site gerar timestamps locais em uma velocidade mais rápida que a de outros sites. Nesse caso, o contador lógico do site rápido será maior do que o dos outros sites. Portanto, todos os timestamps gerados pelo site rápido serão maiores do que aqueles gerados por outros sites. O que precisamos é de um mecanismo para garantir que os timestamps locais sejam gerados de forma justa pelo sistema. Definimos dentro de cada site S_i um **clock lógico** (LC_i), que gera o timestamp local exclusivo. O clock lógico pode ser implementado como um contador que é incrementado após um novo timestamp local ser gerado. Para garantir que os diversos clocks lógicos sejam sincronizados, exigimos que um site S_i avance seu clock lógico sempre que uma transação T_i com timestamp <x,y> visitar esse site e x for maior que o valor atual de LC_i. Nesse caso, o site S_i avança seu clock lógico para o valor $x + 1$.

Se o clock do sistema for usado para gerar timestamps, então elas serão atribuídas justamente, desde que nenhum site tenha um clock do sistema que funcione mais rápido ou mais lento. Como os clocks podem não ser perfeitamente precisos, uma técnica semelhante àquela para clocks lógicos precisa ser usada para garantir que nenhum clock fique à frente ou atrás de outro clock.

Replicação com graus de consistência fracos

Muitos bancos de dados comerciais hoje admitem replicação, que pode assumir uma de várias formas. Com a **replicação mestre-escravo**, o banco de dados permite atualizações em um site primário e propaga automaticamente as atualizações para as réplicas em outros sites. As transações podem ler as réplicas em outros sites, mas não têm permissão para atualizá-las.

Um recurso importante de tal replicação é que as transações não obtêm bloqueios em sites remotos. Para garantir que as transações executando nos sites de réplica tenham uma visão consistente (mas talvez desatualizada) do banco de dados, a réplica deverá refletir um **snapshot consistente com a transação** dos dados no site primário; ou seja, a réplica deverá refletir todas as atualizações das transações até alguma transação na ordem de seriação, e não deverá refletir quaisquer atualizações de outras transações na ordem de seriação.

O banco de dados pode ser configurado para propagar as atualizações imediatamente após elas ocorrerem no site primário, ou para propagar as atualizações apenas periodicamente.

A replicação mestre-escravo é particularmente útil para distribuir informações, por exemplo, de um escritório central para os escritórios das filiais de uma organização. Outro uso para essa forma de replicação é na criação de uma cópia do banco de dados para executar grandes consultas, de modo que as consultas não interfiram com as transações. As atualizações devem ser propagadas periodicamente – a cada noite, por exemplo – de modo que a propagação da atualização não interfira com o processamento da consulta.

O sistema de banco de dados Oracle admite uma instrução **create snapshot**, que pode criar uma cópia snapshot de uma relação consistente com a transação, ou de um conjunto de relações, em um site remoto. Ele também admite o refresh do snapshot, que pode ser feito recalculando o snapshot ou atualizando-o de modo incremental. O Oracle admite o refresh automático, seja de forma contínua ou em intervalos periódicos.

Com a **replicação multimestre** (também chamada **replicação de atualização em qualquer lugar**), as atualizações são permitidas em qualquer réplica de um item de dados e são propagadas automaticamente a todas as réplicas. Esse modelo é o modelo básico usado para gerenciar réplicas nos bancos de dados distribuídos. As transações atualizam a cópia local e o sistema atualiza outras réplicas de forma transparente.

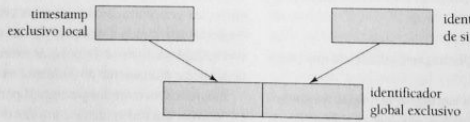


Figura 22.2 Geração de timestamps exclusivos.

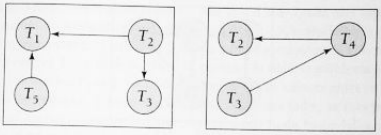


Figura 22.3 Gráficos de espera locais.

Um modo de atualizar as réplicas é aplicar a atualização imediata com o commit de duas fases, usando uma das técnicas de controle de concorrência distribuído que já vimos. Muitos sistemas de banco de dados utilizam o protocolo parcial, em que as escritas precisam bloquear e atualizar todas as réplicas e as leituras bloqueiam e lêem qualquer réplica, como sua técnica de controle de concorrência.

Muitos sistemas de banco de dados oferecem uma forma alternativa de atualização: eles atualizam em um site, com a **propagação lenta** das atualizações para outros sites, em vez de aplicar imediatamente as atualizações a todas as réplicas como parte de uma transação que realiza atualização. Os esquemas baseados na propagação lenta permitem que o processamento da transação (incluindo atualizações) prossiga mesmo que um site esteja desconectado da rede, melhorando assim a disponibilidade, mas, infelizmente, fazem isso ao custo da consistência. Uma de duas técnicas normalmente é seguida quando a propagação lenta é usada:

- As atualizações nas réplicas são traduzidas para atualizações em um site primário, que são então propagadas lentamente a todas as réplicas.

  Essa técnica garante que as atualizações em um item sejam ordenadas em série, embora os problemas de seriação possam ocorrer, visto que as transações podem ler um valor antigo de algum outro item de dados e usá-lo para realizar uma atualização.
- As atualizações são realizadas em qualquer réplica e propagadas a todas as outras réplicas.

  Essa técnica pode causar ainda mais problemas, pois o mesmo item de dados pode ser atualizado simultaneamente em vários sites.

Alguns conflitos devido à falta de controle de concorrência distribuído podem ser detectados quando as atualizações são propagadas para outros sites (veremos como na seção "Desconectividade e consistência" do Capítulo 24), mas resolver o conflito envolve reverter as transações confirmadas, e a durabilidade das transações confirmadas, portanto, não é garantida. Além do mais, a intervenção humana pode ter de lidar com conflitos. Esses esquemas, portanto, deverão ser evitados ou usados com cuidado.

Tratamento de impasse

Os algoritmos de prevenção de impasse e detecção de impasse no Capítulo 16 podem ser usados em um sistema distribuído, desde que sejam feitas algumas modificações. Por exemplo, podemos usar o protocolo de árvore definindo uma árvore *global* entre os itens de dados do sistema. De modo semelhante, a técnica de ordenação de timestamp poderia ser aplicado diretamente a um ambiente distribuído, como vimos na seção "Timestamp".

A prevenção de impasse pode resultar em espera e rollback desnecessários. Além do mais, certas técnicas de prevenção de impasse podem exigir que mais sites estejam envolvidos na execução de uma transação que seria feita de outra forma.

Se permitirmos que os impasses ocorram e contarmos com a detecção do impasse, o problema principal em um sistema distribuído é decidir como manter o gráfico de espera. Técnicas comuns para lidar com essa questão exigem que cada site mantenha um **gráfico de espera local**. Os nós do gráfico correspondem a todas as transações (locais e também não locais) que atualmente estão mantendo ou solicitando qualquer um dos itens locais a esse site. Por exemplo, a Figura 22.3 representa um sistema consistindo em dois sites, cada um mantendo seu gráfico de espera local. Observe que as transações T_2 e T_3 aparecem nos dois gráficos, indicando que as transações solicitaram itens nos dois sites.

Esses gráficos de espera locais são construídos da maneira normal para transações locais e itens de dados. Quando uma transação Ti no site S_1 precisa de um recurso no site S_2, ela envia uma mensagem de solicitação ao site S_2. Se o recurso for mantido pela transação Tj, o sistema insere uma aresta $T_i \rightarrow T_j$ no gráfico de espera local do site S_2.

Claramente, se qualquer gráfico de espera local tiver um ciclo, o impasse terá ocorrido. Por outro lado, o fato de que não existem ciclos em qualquer um dos gráficos de espera

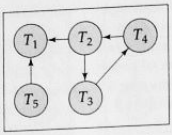


Figura 22.4 Gráfico de espera global para a Figura 22.3.

locais não significa que não existem impasses. Para ilustrar esse problema, consideramos os gráficos de espera locais da Figura 22.3. Cada gráfico de espera é acíclico; apesar disso, existe um impasse no sistema porque a *união* dos gráficos de espera locais contém um ciclo. Esse gráfico aparece na Figura 22.4.

Na técnica de **detecção de impasse centralizada**, o sistema constrói e mantém um **gráfico de espera global** (a união de todos os gráficos locais) em um *único* site: o coordenador de detecção de impasse. Como há atraso de comunicação no sistema, temos de distinguir entre dois tipos de gráficos de espera. O gráfico *real* descreve o estado real mas desconhecido do sistema em qualquer instância no tempo, como seria visto por um observador onisciente. O gráfico *construído* é uma aproximação gerada pelo controlador durante a execução do algoritmo do controlador. Obviamente, o controlador precisa gerar o gráfico construído de modo que, sempre que o algoritmo de detecção for invocado, os resultados informados sejam corretos. *Correto* significa, neste caso, que se houver um impasse, ele será informado prontamente, e se o sistema informar um impasse, ele realmente estará em um estado de impasse.

O gráfico de espera global pode ser reconstruído ou atualizado sob estas condições:

- Sempre que uma nova aresta é inserida ou removida de um dos gráficos de espera.
- Periodicamente, quando um certo número de mudanças tiver ocorrido em um gráfico de espera local.
- Sempre que o coordenador precisar invocar o algoritmo de detecção de ciclo.

Sempre que o coordenador invoca o algoritmo de detecção de impasse, ele pesquisa seu gráfico global. Se encontrar um ciclo, ele seleciona uma vítima a ser revertida. O coordenador precisa notificar a todos os sites de que uma transação em particular foi selecionada como vítima. Os sites, por sua vez, revertem a transação da vítima.

Esse esquema pode produzir descartes desnecessários se:

- **Ciclos falsos** existirem no gráfico de espera global. Como ilustração, considere um instantâneo do sistema representado pelos gráficos de espera locais da Figura 22.5. Suponha que T_2 libere o recurso que está mantendo no site S_1, resultando na exclusão da aresta $T_1 \rightarrow T_2$

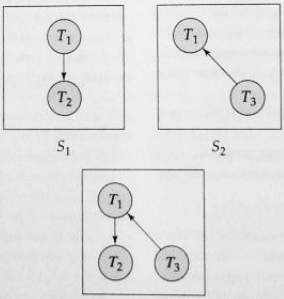


Figura 22.5 Ciclos falsos do gráfico de espera global.

em S_1. A transação T_2, então, solicita um recurso mantido por T_3 no site S_2, resultando no acréscimo da aresta $T_2 \rightarrow T_3$ em S_2. Se a mensagem `insert` $T_2 \rightarrow T_3$ de S_2 chegar antes da mensagem `remove` $T_1 \rightarrow T_2$ de S_1, o coordenador pode descobrir o ciclo falso $T_1 \rightarrow T_2 \rightarrow T_3$ depois do `insert` (mas antes do `remove`). A recuperação do impasse pode ser iniciada, embora nenhum impasse tenha ocorrido.

Observe que a situação do ciclo falso poderia não ocorrer sob o bloqueio de duas fases. A probabilidade de ciclos falsos normalmente é tão baixa que não causa um problema sério no desempenho.

- Um *impasse* realmente ocorreu e uma vítima foi selecionada, enquanto uma das transações foi abortada por motivos não relacionados ao impasse. Por exemplo, suponha que o site S_1 da Figura 22.3 decida abortar T_2. Ao mesmo tempo, o coordenador descobriu um ciclo e selecionou T_3 como vítima. Tanto T_2 quanto T_3 agora são revertidos, embora somente T_2 tivesse de ser revertido.

A detecção de impasse pode ser feita de um modo distribuído, com vários sites assumindo partes da tarefa, em vez de ser feita em um único site. Porém, tais algoritmos são mais complicados e mais dispendiosos. Veja referências a esses algoritmos nas notas bibliográficas deste capítulo.

Disponibilidade

Um dos objetivos no uso de bancos de dados distribuídos é a alta **disponibilidade**; ou seja, o banco de dados precisa funcionar em quase todo o tempo. Em particular, como as falhas são mais prováveis em grandes sistemas distribuídos, um banco de dados distribuído precisa continuar funcionando mesmo quando existem vários tipos de falhas. A capacidade de continuar funcionando mesmo durante as falhas é conhecida como **robustez**.

Para que um sistema distribuído seja robusto, ele precisa *detectar* falhas, *reconfigurar* o sistema para que a computação possa continuar e *recuperar-se* quando um processador ou um enlace for reparado.

Os diferentes tipos de falhas são tratados de diferentes maneiras. Por exemplo, a perda de mensagem é tratada pela retransmissão. A retransmissão repetida de uma mensagem por um enlace, sem recebimento de uma confirmação, normalmente é um sintoma de uma falha no enlace. A rede normalmente tenta encontrar uma rota alternativa para a mensagem. Deixar de encontrar tal rota normalmente é um sintoma de partição da rede.

Porém, geralmente não é possível diferenciar claramente entre falha do site e partição de rede. O sistema normalmente pode detectar que uma falha ocorreu, mas pode não ser capaz de identificar o tipo de falha. Por exemplo, suponha que o site S_1 não seja capaz de se comunicar com S_2. Pode ser que S_2 tenha falhado. Porém, outra possibilidade é de que o enlace entre S_1 e S_2 tenha falhado, resultando na partição da rede. O problema é resolvido parcialmente pelo uso de vários enlaces entre os sites, de modo que, mesmo que um enlace falhe, os sites permaneçam conectados. Porém, ainda pode haver falha em vários enlaces, de modo que existem situações em que não podemos ter certeza se houve uma falha do site ou uma falha de partição de rede.

Suponha que o site S_1 tenha descoberto que houve uma falha. Então, ele precisa iniciar um procedimento que permita que o sistema seja reconfigurado e continue com o modo de operação normal.

- Se as transações estavam ativas em um site com falha/inacessível no momento da falha, essas transações deverão ser abortadas. É desejável abortar tais transações prontamente, pois elas podem manter bloqueios sobre os dados em sites que ainda estão ativos; esperar que o site com falha/inacessível se torne acessível novamente pode impedir outras transações nos sites que estão operando.

  Porém, em alguns casos, quando objetos de dados são replicados, pode ser possível prosseguir com leituras e atualizações mesmo quando algumas réplicas estão inacessíveis. Nesse caso, quando um site com falha se recupera, se ele teve réplicas de qualquer objeto de dados, terá de obter os valores atuais desses objetos de dados e garantir que receba todas as atualizações futuras. Resolvemos essa questão na próxima seção.
- Se os dados replicados estão armazenados em um site com falha/inacessível, o catálogo deve ser atualizado de modo que as consultas não referenciem a cópia no site que falhou. Quando um site se junta novamente, deve-se ter o cuidado de garantir que os dados no site estejam consistentes, conforme veremos na seção "Reintegração do site".
- Se um site que falhou for um servidor central para algum subsistema, uma *eleição* terá de ser mantida para determinar o novo servidor (ver seção "Seleção do coordenador"). Alguns exemplos de servidores centrais incluem um servidor de nomes, um coordenador de concorrência ou um detector global de impasse.

Visto que, em geral, não é possível distinguir entre falhas de enlace de rede e falhas do site, qualquer esquema de reconfiguração precisa ser criado para funcionar corretamente no caso de um particionamento da rede. Em particular, estas situações precisam ser evitadas:

- Dois ou mais servidores centrais são eleitos em partições distintas.
- Mais de uma partição atualiza um item de dados replicado.

Técnica baseada na maioria

A técnica de controle de concorrência distribuído baseada na maioria, da seção "Protocolo da maioria", pode ser modificada para atuar apesar das falhas. Nessa técnica, cada objeto de dado armazena consigo um número de versão para detectar quando foi escrito por último. Sempre que uma transação escreve um objeto, ela também atualiza o número de versão desta maneira:

- Se o objeto de dados *a* for replicado em *n* sites diferentes, então uma mensagem de solicitação de bloqueio precisa ser enviada a mais de metade dos *n* sites em que *a* está armazenado. A transação nunca opera sobre *a* até que tenha obtido com sucesso um bloqueio sobre uma maioria das réplicas de *a*.
- Operações de leitura examinam todas as réplicas em que um bloqueio foi obtido e lêem o valor da réplica que tem o número de versão mais alto. (Opcionalmente, elas também podem escrever esse valor de volta às réplicas com menores números de versão.) As escritas lêem todas as réplicas, assim como as leituras, para encontrar o número de versão mais alto (essa etapa normalmente teria sido realizada antes na transação, por uma leitura, e o resultado pode ser reutilizado). O novo número de versão é um a mais que o número de versão mais alto. A operação de escrita escreve todas as réplicas sobre as quais obteve bloqueios e define o número de versão em todas as réplicas para o novo número de versão.

As falhas durante uma transação (sejam partições de rede ou falhas de site) podem ser toleradas desde que (1) os sites disponíveis no commit contenham a maioria das réplicas de todos os objetos escritos e (2) durante as leituras, a maioria das réplicas seja lida para encontrar os números de versão. Se esses requisitos forem violados, a transação terá de ser abortada. Desde que os requisitos sejam satisfeitos, o protocolo commit de duas fases pode ser usado, como sempre, nos sites que estão disponíveis.

Nesse esquema, a reintegração é trivial; nada precisa ser feito. Isso porque as escritas teriam atualizado a maioria das réplicas, enquanto as leituras lerão a maioria das réplicas e encontrarão pelo menos uma réplica que tem a versão mais recente.

A técnica de numeração de versão usada com o protocolo da maioria também pode ser usada para fazer com que o protocolo de consenso de quórum funcione na presença de falhas. Deixamos os detalhes (simples) para o leitor. Porém, o perigo de falhas que impeçam o sistema de processar transações aumenta se alguns sites receberem pesos mais altos.

Técnica ler em um, escrever em todos os disponíveis

Como um caso especial de consenso de quórum, podemos empregar o protocolo parcial oferecendo pesos unitários a todos os sites, definindo o quórum comum para 1, e definindo o quórum de escrita para *n* (todos os sites). Nesse caso especial, não é preciso usar números de versão; porém, se até mesmo um único site contendo um item de dados falhar, nenhuma escrita no item poderá prosseguir, pois o quórum de escrita não estará disponível. Esse protocolo é chamado de **ler em um, escrever em todos**, pois todas as réplicas precisam ser escritas.

Para permitir que o trabalho prossiga no caso de falhas, gostaríamos de poder usar um protocolo **ler em um, escrever em todos os disponíveis**. Nessa técnica, uma operação de leitura prossegue como no esquema **ler em um, escrever em todos**: qualquer réplica disponível pode ser lida, e um bloqueio de leitura é obtido nessa réplica. Uma operação de escrita é entregue a todas as réplicas; e bloqueios de escrita são adquiridos sobre todas as réplicas. Se um site estiver parado, o gerenciador de transação prossegue sem esperar que o site se recupere.

Embora essa técnica pareça ser muito atraente, existem várias complicações. Em particular, a falha de comunicação temporária pode fazer com que um site pareça estar indisponível, resultando em uma escrita não sendo realizada, mas quando o enlace é restaurado, o site não está ciente de que precisa realizar algumas ações de reintegração para acompanhar as escritas que perdeu. Além disso, se a rede for particionada, cada partição pode prosseguir atualizando o mesmo item de dados, acreditando que os sites nas outras partições estão todos mortos.

O esquema de ler um, escrever todos os disponíveis pode ser usado se nunca houver qualquer particionamento de rede, mas pode resultar em inconsistências no caso de partições de rede.

Reintegração do site

A reintegração de um site ou enlace reparado ao sistema exige cuidado. Quando um site que falhou se recupera, ele precisa iniciar um procedimento para atualizar suas tabelas do sistema de modo a refletir as mudanças feitas enquanto ele estava parado. Se o site tivesse réplicas de quaisquer itens de dados, ele teria de obter os valores atuais desses itens de dados e garantir que receberia todas as atualizações futuras. A reintegração de um site é mais complicada do que pode parecer à primeira vista, pois pode haver atualizações nos itens de dados processados durante o tempo em que o site está se recuperando.

Uma solução fácil é interromper o sistema inteiro temporariamente enquanto o site que falhou se junta nova-

mente. Porém, na maioria das aplicações, essa parada temporária é inaceitável. Foram desenvolvidas técnicas para permitir que os sites que falharam sejam reintegrados enquanto as atualizações concorrentes aos itens de dados prosseguem simultaneamente. Antes que um bloqueio de leitura ou escrita seja concedido sobre qualquer item de dados, o site precisa garantir que alcançou todas as atualizações ao item de dados. Se um enlace que falhou se recuperar, duas ou mais partições poderão ser unidas novamente. Como um particionamento da rede limita as operações permitidas por alguns ou todos os sites, todos os sites deverão ser informados prontamente da recuperação do enlace. Veja mais informações sobre recuperação em sistemas distribuídos nas notas bibliográficas deste capítulo.

Comparação com o backup remoto

Os sistemas de backup remoto, que estudamos na seção "Sistemas de backup remoto" do Capítulo 17, e a replicação nos bancos de dados distribuídos são duas técnicas alternativas para oferecer alta disponibilidade. A principal diferença entre os dois esquemas é que, com os sistemas de backup remoto, ações como controle de concorrência e recuperação são realizadas em um único site, e somente os dados e os registros de log são replicados no outro site. Em particular, os sistemas de backup remoto ajudam a evitar o commit de duas fases e suas sobrecargas resultantes. Além disso, as transações precisam contatar apenas um site (o site primário) e, assim, evitam a sobrecarga de executar o código da transação em vários sites. Assim, os sistemas de backup remoto oferecem uma técnica para a alta disponibilidade de menor custo que a replicação.

Por outro lado, a replicação pode oferecer maior disponibilidade, tendo várias réplicas disponíveis e usando o protocolo da maioria.

Seleção do coordenador

Vários dos algoritmos que apresentamos exigem o uso de um coordenador. Se o coordenador falhar devido a uma falha do site em que reside, o sistema só pode continuar a execução iniciando um novo coordenador em outro site. Um modo de continuar a execução é manter um backup ao coordenador, que está pronto para assumir a responsabilidade se o coordenador falhar.

Um **coordenador backup** é um site que, além de outras tarefas, mantém informações suficientes localmente para permitir que assuma o papel de coordenador com o mínimo de interrupção no sistema distribuído. Todas as mensagens direcionadas ao coordenador são recebidas tanto pelo coordenador quanto pelo seu backup. O coordenador backup executa os mesmos algoritmos e mantém a mesma informação de estado interna (como a tabela de bloqueio, para um coordenador de concorrência) do coordenador real. A única diferença na função entre o coordenador e seu backup é que o backup não toma qualquer ação que afete outros sites. Essas ações são deixadas para o coordenador real.

No evento de o coordenador backup detectar a falha do coordenador real, ele assume o papel de coordenador. Como o backup tem toda a informação disponível a ele que o coordenador que falhou tinha, o processamento pode continuar sem interrupção.

A principal vantagem da técnica de backup é a capacidade de continuar o processamento imediatamente. Se um backup não estivesse pronto para assumir a responsabilidade do coordenador, um coordenador recém-apontado teria de buscar informações de todos os sites no sistema de modo que pudesse executar as tarefas de coordenação. Freqüentemente, a única fonte das informações requisitadas é o coordenador que falhou. Nesse caso, pode ser necessário abortar várias (ou todas as) transações ativas e reiniciá-las sob o controle do novo coordenador.

Assim, a técnica do coordenador backup evita uma grande quantidade de atraso enquanto o sistema distribuído se recupera de uma falha do coordenador. A desvantagem é a sobrecarga da execução duplicada das tarefas do coordenador. Além do mais, um coordenador e seu backup precisam se comunicar regularmente para garantir que suas atividades sejam sincronizadas.

Resumindo, a técnica do coordenador backup gera trabalho extra durante o processamento normal para permitir a recuperação rápida de uma falha do coordenador.

Na ausência de um coordenador backup designado, ou para lidar com múltiplas falhas, um novo coordenador pode ser escolhido dinamicamente pelos sites que estão ativos. Os **algoritmos de eleição** permitem que os sites escolham o site para o novo coordenador de uma maneira descentralizada. Os algoritmos de eleição exigem que um número de identificação exclusivo seja associado a cada site ativo no sistema.

O **algoritmo bully** para a eleição funciona da seguinte maneira. Para manter a notação e a discussão simples, considere que o número de identificação do site S_i seja i e que o coordenador escolhido sempre seja o site ativo com o maior número de identificação. Logo, quando um coordenador falha, o algoritmo precisa eleger o site ativo que tem o maior número de identificação. O algoritmo precisa enviar esse número a cada site ativo no sistema. Além do mais, o algoritmo precisa oferecer um mecanismo pelo qual um site se recuperando de uma falha possa identificar o coordenador atual. Suponha que o site S_i envie uma solicitação que não é respondida pelo coordenador dentro do intervalo de tempo pré-especificado T. Nessa situação, considera-se que o coordenador falhou, e S_i tenta se eleger como site para o novo coordenador.

O site S_i envia uma mensagem de eleição para cada site que possui um número de identificação mais alto. O site S_i, então, espera por um intervalo de tempo T, por uma resposta de qualquer um desses sites. Se ele não receber resposta dentro do tempo T, considera que todos os sites com números maiores do que i falharam, se elege como site para o novo coordenador e envia uma mensagem para informar a todos os sites ativos com números de identificação menores que i de que é o site em que o novo coordenador reside.

Se S_i receber uma resposta, ele começará um intervalo de tempo T', para receber uma mensagem informando de que um site com um número de identificação maior foi eleito. (Algum outro site está se elegendo como coordenador e deve informar os resultados dentro do tempo T'.) Se S_i não receber uma mensagem dentro de T', então ele considera que o site com um número maior falhou, e o site S_i reinicia o algoritmo.

Depois que o site que falhou se recuperar, ele imediatamente inicia a execução do mesmo algoritmo. Se não houver sites ativos com números maiores, o site recuperado força todos os sites com números menores a permitirem que ele se torne o site coordenador, mesmo que haja um coordenador atualmente ativo com um número inferior. É por esse motivo que o algoritmo é chamado de algoritmo *bully* (valentão). Se a rede for particionada, o algoritmo bully elege um coordenador separado em cada partição; para garantir que no máximo um coordenador seja eleito, os sites vencedores devem adicionalmente verificar se a maioria dos sites está em sua partição.

Processamento de consulta distribuído

No Capítulo 14, vimos que existem diversos métodos para calcular a resposta a uma consulta. Examinamos várias técnicas para escolher uma estratégia a fim de processar uma consulta que minimiza a quantidade de tempo gasto ao calcular a resposta. Para sistemas centralizados, o principal critério para medir o custo de determinada estratégia é o número de acessos ao disco. Em um sistema distribuído, temos de levar em conta várias outras questões, incluindo

- O custo de transmissão de dados pela rede
- O ganho de potencial no desempenho de ter vários sites processando partes da consulta em paralelo

O custo relativo da transferência de dados pela rede e da transferência de dados de e para o disco varia muito, dependendo do tipo de rede e da velocidade dos discos. Assim, em geral, não podemos focalizar somente os custos de disco ou os custos da rede. Em vez disso, temos de encontrar um bom meio-termo entre os dois.

Transformação de consulta

Considere uma consulta extremamente simples: "Encontrar todas as tuplas na relação *conta*". Embora a consulta seja simples – na verdade, trivial –, seu processamento não é trivial, pois a relação *conta* pode ser fragmentada, replicada ou ambos, como vimos na seção "Armazenamento de dados distribuído". Se a relação *conta* for replicada, temos uma escolha de réplica a fazer. Se nenhuma réplica for fragmentada, escolhemos a réplica para a qual o custo de transmissão é o menor. No entanto, se uma réplica for fragmentada, a escolha não é tão fácil de fazer, pois precisamos calcular várias junções ou uniões de modo a reconstruir a relação *conta*. Nesse caso, o número de estratégias para nosso exemplo simples pode ser grande. A otimização da consulta pela enumeração completa de todas as estratégias alternativas pode não ser prática em tais situações.

A transparência da fragmentação implica que um usuário pode escrever uma consulta como

$$\sigma_{nome\_agência = \text{"Hillside"}}(conta)$$

Visto que *conta* é definida como

$$conta_1 \cup conta_2$$

a expressão que resulta do esquema de tradução de nome é

$$\sigma_{nome\_agência = \text{"Hillside"}}(conta_1 \cup conta_2)$$

Usando as técnicas de otimização de consulta do Capítulo 14, podemos simplificar a expressão anterior automaticamente. O resultado é a expressão

$$\sigma_{nome\_agência = \text{"Hillside"}}(conta_1) \cup \sigma_{nome\_agência = \text{"Hillside"}}(conta_2)$$

que inclui duas subexpressões. A primeira envolve apenas $conta_1$, e por isso pode ser avaliada no site Hillside. A segunda envolve apenas $conta_2$, e portanto pode ser avaliada no site Valleyview. Há uma outra otimização que pode ser feita na avaliação de

$$\sigma_{nome\_agência = \text{"Hillside"}}(conta_1)$$

Visto que $conta_1$ tem apenas tuplas pertencentes à agência Hillside, podemos eliminar a operação de seleção. Na avaliação de

$$\sigma_{nome\_agência = \text{"Hillside"}}(conta_2)$$

podemos aplicar a definição do fragmento $conta_2$ para obter

$$\sigma_{nome\_agência = \text{"Hillside"}}(\sigma_{nome\_agência = \text{"Valleyview"}}(conta))$$

Essa expressão é o conjunto vazio, independente do conteúdo da relação *conta*. Assim, nossa estratégia final é que o site Hillside retorne $conta_1$ como resultado da consulta.

Processamento de junção simples

Como vimos no Capítulo 14, uma decisão importante na seleção de uma estratégia de processamento de consulta é escolher uma estratégia de junção. Considere a seguinte expressão da álgebra relacional:

$$conta \bowtie depositante \bowtie agência$$

Considere que as três relações não são replicadas nem fragmentadas, e que *conta* é armazenada no site S_1, *depositante* em S_2, e *agência* em S_3. Considere que S_1 indica o site em que a consulta foi emitida. O sistema precisa produzir o resultado no site S_1. Entre as estratégias possíveis para o processamento dessa consulta estão estas:

- Enviar cópias de todas as três relações para o site S_1. Usando as técnicas do Capítulo 14, escolha uma estratégia para processar a consulta inteira localmente no site S_1.
- Enviar uma cópia da relação *conta* para o site S_2 e calcular $temp_1 = conta \bowtie depositante$ em S_2. Enviar $temp_1$ de S_2 para S_3 e calcular $temp_2 = temp_1 \bowtie agência$ em S_3. Enviar o resultado $temp_2$ para S_1.
- Criar estratégias semelhantes à anterior, com os papéis de S_1, S_2, S_3 trocados.

Nenhuma estratégia é sempre a melhor. Entre os fatores que precisam ser considerados estão o volume dos dados sendo enviados, o custo da transmissão de um bloco de dados entre um par de sites e a velocidade relativa do processamento em cada site. Considere as duas primeiras estratégias listadas. Suponha que os índices presentes em S_2 e S_3 sejam úteis para calcular a junção. Se enviarmos todas as três relações para S_1, precisaríamos recriar esses índices em S_1 ou usar uma estratégia de junção diferente, possivelmente mais dispendiosa. A recriação de índices vincula sobrecarga de processamento extra e acessos extras ao disco. Com a segunda estratégia, uma relação potencialmente grande ($conta \bowtie depositante$) precisa ser enviada de S_2 para S_3. Essa relação repete o nome de um cliente uma vez para cada conta que o cliente possui. Assim, a segunda estratégia pode resultar em transmissão de rede extra, em comparação com a primeira estratégia.

Estratégia de semijunção

Suponha que queremos avaliar a expressão $r_1 \bowtie r_2$, em que r_1 e r_2 são armazenadas nos sites S_1 e S_2, respectivamente.

Considere que os esquemas de r_1 e r_2 sejam R_1 e R_2. Suponha que queremos obter o resultado em S_1. Se houver muitas tuplas de r_2 que não se juntam com qualquer tupla de r_1, então o envio de r_2 para S_1 acarreta o envio de tuplas que deixam de contribuir para o resultado. Queremos remover essas tuplas antes de enviar dados a S_1, particularmente se os custos de rede forem altos.

Uma estratégia possível para conseguir tudo isso é:

1. Calcule $temp_1 \leftarrow \prod_{R_1 \cap R_2}(r_1)$ em S_1.
2. Envie $temp_1$ de S_1 para S_2.
2. Calcule $temp_2 \leftarrow r_2 \bowtie temp_1$ em S_2.
3. Envie $temp_2$ de S_2 para S_1.
4. Calcule $r_1 \bowtie temp_2$ em S_1. A relação resultante é a mesma que $r_1 \bowtie r_2$.

Antes de considerar a eficiência dessa estratégia, vamos verificar se a estratégia calcula a resposta correta. Na etapa 3, $temp_2$ tem o resultado de $r_2 \bowtie \prod_{R_1 \cap R_2}(r_1)$. Na etapa 5, calculamos

$$r_1 \bowtie r_2 \bowtie \prod_{R_1 \cap R_2}(r_1)$$

Como a junção é associativa e cumulativa, podemos reescrever essa expressão como

$$(r_1 \bowtie \prod_{R_1 \cap R_2}(r_1)) \bowtie r_2$$

Como $r_1 \bowtie \prod_{R_1 \cap R_2}(r_1) = r_1$, a expressão é, na verdade, igual a $r_1 \bowtie r_2$, a expressão que estamos tentando avaliar.

Essa estratégia é particularmente vantajosa quando relativamente poucas tuplas de r_2 contribuem para a junção. Essa situação provavelmente ocorrerá se r_1 for o resultado de uma expressão da álgebra relacional envolvendo seleção. Nesse caso, $temp_2$ pode ter muito menos tuplas do que r_2. As economias de custo da estratégia resultam de ter de enviar apenas $temp_2$, em vez de todos os r_2, para S_1. O custo adicional é incorrido no envio de $temp_1$ para S_2. Se uma fração suficientemente pequena das tuplas em r_2 contribuir com a junção, a sobrecarga do envio de $temp_1$ será dominada pelas economias de envio apenas de uma fração das tuplas em r_2.

Essa estratégia é chamada **estratégia de semijunção**, depois do operador de semijunção da álgebra relacional, indicada por $\ltimes$. A semijunção de r_1 com r_2, indicada por $r_1 \ltimes r_2$, é

$$\prod_{R_1}(r_1 \bowtie r_2)$$

Assim, $r_1 \ltimes r_2$ seleciona aquelas tuplas da relação r_1 que contribuíram para $r_1 \bowtie r_2$. Na etapa 3, $temp_2 = r_2 \ltimes r_1$.

Para junções de várias relações, essa estratégia pode ser estendida para uma série de etapas de semijunção. Uma

quantidade substancial de teoria foi desenvolvida com relação ao uso de semijunções para otimização de consulta. Parte dessa teoria é referenciada nas notas bibliográficas.

Estratégias de junção que exploram o paralelismo

Considere uma junção de quatro relações:

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

onde a relação r_i é armazenada no site S_i. Suponha que o resultado tenha de ser apresentado no site S_1. Existem muitas estratégias possíveis para a avaliação paralela. (Estudamos a questão de processamento paralelo de consultas com detalhes no Capítulo 21.) Em uma estratégia desse tipo, r_1 é enviada a S_2, e $r_1 \bowtie r_2$ computada em S_2. Ao mesmo tempo, r_3 é enviada a S_4, e $r_3 \bowtie r_4$ computada em S_4. O site S_2 pode enviar tuplas de $(r_1 \bowtie r_2)$ a S_1 enquanto são produzidas, em vez de esperar que a junção inteira seja computada. De modo semelhante, S_4 pode enviar tuplas de $(r_3 \bowtie r_4)$ a S_1. Quando as tuplas de $(r_1 \bowtie r_2)$ e $(r_3 \bowtie r_4)$ chegarem em S_1, a computação de $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ pode começar, com a técnica de junção canalizada da seção "Algoritmos de avaliação para a canalização" do Capítulo 13. Assim, a computação do resultado final da junção em S_1 pode ser feita em paralelo com a computação de $(r_1 \bowtie r_2)$ em S_2 e com a computação de $(r_3 \bowtie r_4)$ em S_4.

Bancos de dados distribuídos heterogêneos

Muitas aplicações de banco de dados novas exigem dados de diversos bancos de dados preexistentes localizados em uma coleção heterogênea de ambientes de hardware e software. A manipulação de informações localizadas em um banco de dados distribuído heterogêneo requer uma camada de software adicional em cima dos sistemas de banco de dados existentes. Essa camada de software é chamada de **sistema multibanco de dados**. Os sistemas de banco de dados locais podem empregar diferentes modelos lógicos e linguagens de definição e manipulação de dados, e podem diferir em seu controle de concorrência e mecanismos de gerenciamento de transação. Um sistema multibanco de dados cria a ilusão de integração lógica de banco de dados sem exigir a integração física do banco de dados.

A integração total de sistemas heterogêneos a um banco de dados distribuído homogêneo normalmente é difícil ou impossível:

- **Dificuldades técnicas.** O investimento em programas de aplicação com base nos sistemas de banco de dados existentes pode ser imenso, e o custo de conversão dessas aplicações pode ser proibitivo.
- **Dificuldades organizacionais.** Mesmo que a integração seja *tecnicamente* possível, ela pode não ser *politicamente* possível, pois os sistemas de banco de dados existentes pertencem a diferentes corporações ou organizações. Em tais casos, é importante para um sistema multibanco de dados permitir que os sistemas de banco de dados locais retenham um alto grau de **autonomia** em relação ao banco de dados local e transações executando sobre esses dados.

Por esses motivos, os sistemas multibanco de dados oferecem vantagens significativas que pesam mais que sua sobrecarga. Nesta seção, oferecemos uma visão geral dos desafios encarados na construção de um ambiente multibanco de dados do ponto de vista da definição de dados e do processamento de consulta. A seção "Gerenciamento de transações em bancos de dados múltiplos" do Capítulo 25 oferece uma visão geral das questões de gerenciamento de transação em multibancos de dados.

Visão unificada dos dados

Cada sistema de gerenciamento de banco de dados local pode usar um modelo de dados diferente. Por exemplo, alguns podem empregar o modelo relacional, enquanto outros podem empregar modelos de dados mais antigos, como o modelo de rede (ver Apêndice A) ou o modelo hierárquico (ver Apêndice B).

Como o sistema multibanco de dados deverá oferecer a ilusão de um único sistema de banco de dados integrado, um modelo de dados comum precisa ser usado. Uma escolha normalmente usada é o modelo relacional, com a SQL como linguagem de consulta comum. Na verdade, existem vários sistemas disponíveis hoje para permitir consultas SQL em um sistema de gerenciamento de banco de dados não relacional.

Outra dificuldade é a provisão de um esquema conceitual comum. Cada sistema local oferece seu próprio esquema conceitual. O sistema multibanco de dados precisa integrar esses esquemas separados em um esquema comum. A integração do esquema é uma tarefa complicada, principalmente por causa da heterogeneidade semântica.

A integração do esquema não é simplesmente a tradução direta entre diferentes linguagens de definição. Os mesmos nomes de atributo podem aparecer em diferentes bancos de dados locais, mas com diferentes significados. Os tipos de dados usados em um sistema podem não ser admitidos por outros sistemas, e a tradução entre os tipos pode não ser simples. Mesmo para tipos de dados idênticos, os problemas podem surgir pela representação física dos dados: um

sistema pode usar ASCII, outro EBCDIC; as representações em ponto flutuante podem diferir; inteiros podem ser representados por formato *big-endian* ou *little-endian*. No nível semântico, um valor inteiro para o tamanho pode ser polegadas em um sistema e milímetros em outro, criando assim uma situação esquisita, em que a igualdade de inteiros é apenas uma noção aproximada (como sempre acontece para números de ponto flutuante). O mesmo nome pode aparecer em diferentes linguagens em diferentes sistemas. Por exemplo, um sistema baseado nos Estados Unidos pode se referir à cidade "Cologne", enquanto na Alemanha refere-se a ela como "Köln".

Todas essas distinções aparentemente pequenas precisam ser registradas de forma correta no esquema conceitual global comum. As funções de tradução precisam ser fornecidas. Os índices precisam ser anotados para o comportamento dependente do sistema (por exemplo, a ordem de classificação de caracteres não alfanuméricos não é a mesma em ASCII e em EBCDIC). Conforme observamos anteriormente, a alternativa de converter cada banco de dados a um formato comum pode não ser viável sem tornar obsoletos os programas de aplicação existentes.

Processamento de consulta

O processamento da consulta em um banco de dados heterogêneo pode ser complicado. Algumas das questões são:

- Dada uma consulta em um esquema global, a consulta pode ter de ser traduzida para consultas em esquemas locais em cada um dos sites em que a consulta precisa ser executada. Os resultados da consulta precisam ser traduzidos de volta para o esquema global.

  A tarefa é simplificada pela escrita de **wrappers** para cada origem de dados, que oferece uma visão dos dados locais no esquema global. Os wrappers também traduzem consultas no esquema global para consultas no esquema local, e traduzem resultados de volta para o esquema global. Wrappers podem ser fornecidos por sites individuais, ou podem ser escritos separadamente como parte do sistema de multibanco de dados.

  Os wrappers podem ainda ser usados para oferecer uma visão relacional das origens de dados não relacionais, como páginas Web (possivelmente com interfaces de formulários), arquivos puros, bancos de dados hierárquicos e de rede, além de sistemas de diretório.
- Algumas origens de dados podem oferecer apenas capacidades limitadas de consulta; por exemplo, elas podem admitir seleções, mas não junções. Elas podem ainda restringir a forma das seleções, permitindo seleções apenas sobre certos campos; as origens de dados da Web com interfaces de formulário são um exemplo de tais origens de dados. As consultas, portanto, podem ter de ser desmembradas, para serem parcialmente realizadas na origem de dados e parcialmente no site que emite a consulta.
- Em geral, mais de um site pode ter de ser acessado para responder a determinada consulta. As respostas apanhadas dos sites podem ter de ser processadas para remover duplicatas. Suponha que um site contenha tuplas *conta* satisfazendo a seleção *saldo* < 100, enquanto outro contém tuplas *conta* satisfazendo *saldo* > 50. Uma consulta sobre a relação conta inteira exigiria acesso aos dois sites e remoção de respostas duplicadas, resultantes de tuplas com saldo entre 50 e 100, que são replicadas nos dois sites.
- A otimização de consulta global em um banco de dados heterogêneo é difícil, pois o sistema de execução de consulta pode não saber quais custos são de planos de consulta alternativos nos diferentes sites. A solução normal é contar apenas com a otimização de nível local e usar apenas a heurística no nível global.

Sistemas **mediadores** são sistemas que integram várias origens de dados heterogêneas, oferecendo uma visão global integrada dos dados e oferecendo facilidades de consulta na visão global. Ao contrário dos sistemas multibanco de dados com todos os recursos, os sistemas mediadores não se importam com o processamento da transação. (Os termos mediador e multibanco de dados normalmente são usados de uma maneira intercambiável, e os sistemas que são chamados mediadores podem admitir formas limitadas de transações.) O termo **banco de dados virtual** é usado para se referir a sistemas multibanco de dados/mediadores, pois oferecem a aparência de um único banco de dados com um esquema global, embora existam dados em vários sites nos esquemas locais.

Sistemas de diretório

Considere uma organização que queira tornar os dados sobre seus funcionários disponíveis a uma série de pessoas na organização; alguns exemplos dos tipos de dados incluem nome, designação, id-funcionário, endereço, número de telefone, número de fax e assim por diante. Nos dias anteriores aos computadores, as organizações criariam diretórios físicos de funcionários e os distribuiriam pela organização. Ainda hoje, companhias de telefone criam diretórios físicos dos clientes.

Em geral, um diretório é uma listagem de informações sobre alguma classe de objetos, como pessoas. Os diretórios podem ser usados para encontrar informações sobre um objeto específico, ou na direção reversa, para encontrar objetos que atendem a um certo requisito. No mundo dos di-

retórios de telefone físicos (os catálogos telefônicos), aqueles que satisfazem as pesquisas na direção normal são chamados **páginas brancas**, enquanto os diretórios que satisfazem pesquisas na direção contrária são chamados **páginas amarelas**.

No mundo atual interligado por redes, a necessidade de diretórios ainda está presente, e ainda é mais importante. Porém, os diretórios hoje precisam estar disponíveis por uma rede de computadores, em vez de uma forma física (papel).

Protocolos de acesso ao diretório

As informações de diretório podem se tornar disponíveis por meio de interfaces da Web, como fazem muitas organizações, principalmente companhias telefônicas. Essas interfaces são boas para os humanos. Porém, os programas também precisam acessar informações de diretório. Os diretórios podem ser usados para armazenar outros tipos de informações, muito semelhante aos diretórios do sistema de arquivos. Por exemplo, os navegadores Web podem armazenar marcadores pessoais e outras configurações de navegador em um sistema de diretório. Um usuário pode, assim, acessar as mesmas configurações a partir de vários locais, como em casa e no trabalho, sem ter de compartilhar um sistema de arquivos.

Vários **protocolos de acesso a diretório** foram desenvolvidos para oferecer uma forma padronizada de acessar dados em um diretório. O mais utilizado entre eles hoje é o **Lightweight Directory Access Protocol (LDAP)**.

Obviamente, todos os tipos de dados em nossos exemplos podem ser armazenados sem muito trabalho em um sistema de banco de dados e acessados por meio de protocolos como SQL ou ODBC. Logo, a questão é: por que a necessidade de um protocolo especializado para acessar informações de diretório? Existem pelo menos duas respostas para a pergunta:

- Primeiro, os protocolos de acesso ao diretório são protocolos simplificados, que satisfazem um tipo limitado de acesso aos dados. Eles evoluíram em paralelo com os protocolos de acesso ao banco de dados.
- Segundo, e mais importante, os sistemas de diretório oferecem um mecanismo simples para nomear objetos de uma forma hierárquica, semelhante aos nomes de diretório do sistema de arquivos, que podem ser usados em um sistema de diretório distribuído para especificar que informações são armazenadas em cada um dos servidores de diretório. Por exemplo, determinado servidor de diretório pode armazenar informações para os funcionários da Bell Laboratories em Murray Hill, enquanto outro pode armazenar informações para funcionários da Bell Laboratories em Bangalore, dando a ambos os sites autonomia no controle de seus dados locais. O protocolo de acesso ao diretório pode ser usado para obter dados dos dois diretórios, através de uma rede. Mais importante, o sistema de diretório pode ser configurado para encaminhar automaticamente as consultas feitas em um site para outro, sem intervenção do usuário.

Por esses motivos, várias organizações possuem sistemas de diretório para tornar as informações organizacionais disponíveis on-line por meio de um protocolo de acesso ao diretório. As informações em um diretório organizacional podem ser usadas para uma série de propósitos, como descobrir endereço, telefone ou endereços de e-mail das pessoas, descobrir em quais departamentos as pessoas estão e acompanhar hierarquias de departamento. Os diretórios também são usados para autenticar usuários: aplicações podem coletar informações de autenticação, como as senhas dos usuários, e autenticá-los usando o diretório.

Como poderíamos esperar, várias implementações de diretório acham benéfico usar bancos de dados relacionais para armazenar dados, em vez de criar sistemas de armazenamento de uso especial.

LDAP: Lightweight Directory Access Protocol

Em geral, um sistema de diretório é implementado como um ou mais servidores, que atendem a vários clientes. Os clientes usam a interface de programação de aplicação definida pelo sistema de diretório para se comunicarem com os servidores de diretório. Os protocolos de acesso ao diretório também definem um modelo de dados e controle de acesso.

O **protocolo de acesso a diretório X.500**, definido pela International Organization for Standardization (ISO), é um padrão para acessar informações de diretório. Porém, o protocolo é um tanto complexo e não é muito utilizado. O **Lightweight Directory Access Protocol (LDAP)** oferece muitos dos recursos do X.500, mas com menos complexidade, e é muito utilizado. No restante desta seção, esboçaremos o modelo de dados e os detalhes do protocolo de acesso do LDAP.

Modelo de dados LDAP

No LDAP, os diretórios armazenam **entradas**, que são semelhantes aos objetos. Cada entrada precisa ter um **Distinguished Name (DN)**, que a identifica exclusivamente. Um DN, por sua vez, é composto de uma seqüência de **Relative Distinguished Names (RDNs)**. Por exemplo, uma entrada pode ter o seguinte nome distinto:

```
cn=Silberschatz, ou=BellLabs, o=Lucent, c=USA
```

Como você pode ver, o nome distinto neste exemplo é uma combinação de um nome e endereço (organizacional), começando com o nome de uma pessoa, depois informando a unidade organizacional (ou), a organização (o) e o país (c). A ordem dos componentes de um nome distinto reflete a ordem normal do endereço postal, em vez da ordem reversa usada na especificação de nomes de caminho para arquivos. O conjunto de RDNs para um DN é definido pelo esquema do sistema de diretório.

As entradas também podem ter atributos. LDAP oferece tipos binário, de string e de tempo, e adicionalmente os tipos `tel` para números de telefone, e `PostalAddress` para endereços (linhas separadas por um caractere "$"). Ao contrário daqueles no modelo relacional, os atributos são multivalorados como padrão, de modo que é possível armazenar vários números de telefone ou endereços para uma entrada.

LDAP permite a definição de **classes de objetos** com nomes e tipos de atributos. A herança pode ser usada na definição de classes de objetos. Além do mais, as entradas podem ser especificadas para ser de uma ou mais classes de objeto. Não é necessário que haja uma única classe de objeto mais específica à qual uma entrada pertence.

As entradas são organizadas em uma **Directory Information Tree (DIT)**, de acordo com seus nomes distintos. As entradas no nível de folha da árvore normalmente representam objetos específicos. As entradas que são nós internos representam objetos como unidades organizacionais, organizações ou países. Os filhos de um nó possuem um DN contendo todos os RDNs do pai, e um ou mais RDNs adicionais. Por exemplo, um nó interno pode ter um DN `c=USA`, e todas as entradas abaixo dele possuem o valor `USA` para o RDN `c`.

O nome distinto inteiro não precisa ser armazenado em uma entrada; o sistema pode gerar o nome distinto de uma entrada atravessando a DIT a partir da entrada, coletando os componentes `RDN=valor` para criar o nome distinto completo.

As entradas podem ter mais de um nome distinto – por exemplo, uma entrada para uma pessoa em mais de uma organização. Para lidar com esses casos, o nível de folha de uma DIT pode ser um **alias**, que aponta para uma entrada em outro ramo da árvore.

Manipulação de dados

Diferente da SQL, o LDAP não define uma linguagem de definição de dados ou uma linguagem de manipulação de dados. Porém, o LDAP define um protocolo de rede para executar definição e manipulação de dados. Os usuários do LDAP podem usar uma interface de programação de aplicação ou usar ferramentas de diversos fornecedores para realizar a definição e a manipulação de dados. LDAP também define um formato de arquivo chamado **LDAP Data Interchange Format (LDIF)**, que pode ser usado para armazenar e trocar informações.

O mecanismo de consulta no LDAP é muito simples, consistindo em apenas seleções e projeções, sem qualquer junção. Uma consulta precisa especificar o seguinte:

- Uma base – ou seja, um nó dentro de uma DIT – dando seu nome distinto (o caminho da raiz até o nó).
- Uma condição de busca, que pode ser uma combinação Booleana de condições sobre atributos individuais. De modo semelhante, a combinação por caracteres curinga e igualdade aproximada (a definição exata de igualdade aproximada depende do sistema) são admitidas.
- Um escopo, que pode ser apenas a base, a base e seus filhos, ou a subárvore inteira abaixo da base.
- Atributos para retornar.
- Limites sobre o número de resultados e consumo de recursos.

A consulta também pode especificar se irá desreferenciar aliases automaticamente; se as desreferências de alias estiverem desativadas, as entradas de alias poderão ser retornadas como respostas.

Um modo de consultar uma origem de dados LDAP é usando URLs LDAP. Alguns exemplos de URLs LDAP são:

```
ldap:://aura.research.bell-labs.com/o=Lucent,c=USA
ldap:://aura.research.bell-labs.com/o=Lucent,c=USA??sub
?cn=Korth
```

O primeiro URL retorna todos os atributos de todas as entradas no servidor com organização sendo Lucent e país, USA. O segundo URL executa uma consulta de busca (seleção) `cn=Korth` sobre a subárvore do nó com nome distinto `o=Lucent, c=USA`. Os pontos de interrogação no URL separam campos diferentes. O primeiro campo é o nome distinto, aqui, `o=Lucent, c=USA`. O segundo campo, a lista de atributos a retornar, fica vazio, significando o retorno de todos os atributos. O terceiro atributo, `sub`, indica que a subárvore inteira deve ser pesquisada. O último parâmetro é a condição de busca.

Uma segunda forma de consultar um diretório LDAP é usando uma interface de programação de aplicação. A Figura 22.6 mostra uma parte do código C usado para a conexão com um servidor LDAP e execução de uma consulta contra o servidor. O código primeiro abre uma conexão com um servidor LDAP por `ldap_open` e `ldap_bind`. Depois, ele executa uma consulta por `ldap_search_s`. Os argumentos de `ldap_search_s` são o descritor (handle) da conexão LDAP, o DN da base da qual a busca deverá ser feita, o escopo da busca, a condição da busca, a lista de atributos a serem retornados e um atributo chamado `attrsonly`, que, se definido como 1, resultaria apenas no esquema do

```
#include <stdio.h>
#include <ldap.h>
main( ) {
  LDAP *ld;
  LDAPMessage *res, *entry;
  char *dn, *attr, *attrList[] = {"telephoneNumber", NULL};
  BerElement *ptr;
  int vals, i;
  ld = ldap_open("aura.research.bell-labs.com", LDAP_PORT);
  ldap_simple_bind(ld, "avi", "avi-passwd") ;
  ldap_search_s(ld, "o=Lucent, c=USA", LDAP-SCOPE_SUBTREE, "cn=Korth",
        attrList, /*attrsonly*/ 0, &res);
  printf("found %d entries", ldap_count_entries(ld, res));
  for (entry=ldap_first_entry(ld, res); entry != NULL;
                        entry = ldap_next_entry(ld, entry))
  {
      dn = ldap_get_dn(ld, entry);
      printf("dn: %s", dn);
      ldap_memfree(dn);
      for (attr = ldap_irst_attribute(ld, entry, &ptr);
              attr != NULL;
              attr = ldap_next_attribute(ld, entry, ptr))
      {
          printf("%s: ", attr);
          vals = ldap_get_values(ld, entry, attr);
          for (i=0; vals[i] != NULL; i++)
              printf("%s, ", vals[i]);
          ldap_value_free(vals);
      }
  }
  ldap_msgfree(res);
  ldap_unbind(ld);
}
```

resultado sendo retornado, sem quaisquer tuplas reais. O último argumento é um argumento de saída, que retorna o resultado da busca como uma estrutura `LDAPMessage`.

O primeiro loop **for** percorre e imprime cada entrada no resultado, e o segundo loop **for** imprime cada atributo. Observe que uma entrada pode ter vários atributos. Como os atributos em LDAP podem ter múltiplos valores, o terceiro loop **for** imprime cada valor de um atributo. As chamadas `ldap_msgfree` e `ldap_value_free` liberam a memória que foi alocada pelas bibliotecas LDAP. A Figura 22.6 não mostra o código para o tratamento de condições de erro.

A API LDAP também contém funções para criar, atualizar e excluir entradas, além de outras operações sobre a DIT. Cada chamada de função se comporta como uma transação separada; LDAP não admite atomicidade de múltiplas atualizações.

Árvores de diretório distribuídas

As informações sobre uma organização podem ser divididas em várias DITs, cada qual armazenando informações sobre algumas entradas. O **sufixo** de uma DIT é uma sequência de pares `RDN=valor` que identificam qual informação a DIT armazena; os pares são concatenados ao restante do nome distinto gerado pela travessia da entrada até a raiz. Por exemplo, o sufixo de uma DIT pode ser `o=Lucent, c=USA`, enquanto outra pode ter o sufixo `o=Lucent, c=India`. As DITs podem ser separadas organizacional e geograficamente.

Um nó em uma DIT pode conter um **atestado** para outro nó em outra DIT; por exemplo, a unidade organizacional Bell Labs sob `o=Lucent, c=USA` pode ter sua própria DIT, de modo que a DIT para `o=Lucent, c=USA` teria um nó `ou=BellLabs` representando um atestado à DIT para a Bell Labs.

Os atestados são o principal componente que ajuda a organizar uma coleção distribuída de diretórios em um sistema integrado. Quando um servidor recebe uma consulta sobre uma DIT, ele pode retornar um atestado ao cliente, que então emite uma consulta sobre a DIT referenciada. O acesso à DIT referenciada é transparente, prosseguindo sem o conhecimento do usuário. Como alternativa, o pró-

prio servidor pode emitir a consulta à DIT referenciada e retornar os resultados junto com os resultados calculados no local.

O mecanismo de nomeação hierárquico utilizado pelo LDAP ajuda a dividir o controle da informação entre pares de uma organização. A facilidade de atestado, então, ajuda a integrar todos os diretórios de uma organização a um único diretório virtual.

Embora não sendo um requisito do LDAP, as organizações normalmente decidem desmembrar as informações por geografia (por exemplo, uma organização pode manter um diretório para cada local em que a organização possui uma grande presença) ou por estrutura organizacional (por exemplo, cada unidade organizacional, como departamento, mantém seu próprio diretório).

Muitas implementações LDAP admitem a replicação mestre-escravo e multimestre das DITs, embora a replicação não faça parte do padrão LDAP atual, versão 3. O trabalho na padronização da replicação do LDAP está em andamento.

Resumo

- Um sistema de banco de dados distribuído consiste em uma coleção de sites, cada um mantendo um sistema de banco de dados local. Cada site é capaz de processar transações locais: aquelas transações que acessam dados apenas nesse site isolado. Além disso, um site pode participar na execução de transações globais: aquelas transações que acessam dados em vários sites. A execução de transações globais requer a comunicação entre os sites.
- Os bancos de dados distribuídos podem ser homogêneos, em que todos os sites têm um esquema comum e um código do sistema de banco de dados comum, ou heterogêneos, em que os esquemas e os códigos do sistema podem diferir.
- Existem várias questões envolvidas no armazenamento de uma relação no banco de dados distribuído, incluindo a replicação e a fragmentação. É essencial que o sistema evite ao máximo que o usuário precise entender como uma relação foi armazenada.
- Um sistema distribuído pode sofrer dos mesmos tipos de falha que podem afligir um sistema centralizado. Contudo, existem outras falhas com que precisamos lidar em um ambiente distribuído, incluindo a falha de um site, a falha de um enlace, perda de uma mensagem e partição da rede. Cada um desses problemas precisa ser considerado no projeto de um esquema de recuperação distribuído.
- Para assegurar a atomicidade, todos os sites em que uma transação T foi executada combinam com o resultado final da execução. T é confirmada em todos os sites ou abortada em todos os sites. Para assegurar essa propriedade, o coordenador de transação de T precisa executar um protocolo commit. O protocolo commit mais utilizado é o protocolo commit de duas fases.
- O protocolo commit de duas fases pode lidar com o bloqueio, a situação em que o destino de uma transação não pode ser determinado até que um site que falhou (o coordenador) se recupere. Podemos usar o protocolo commit de três fases para reduzir a probabilidade de bloqueio.
- A técnica de mensagem persistente oferece um modelo alternativo para tratar de transações distribuídas. O modelo desmembra uma única transação em partes, que são executadas em diferentes bancos de dados. As mensagens persistentes (que, por garantia, são entregues exatamente uma vez, independente das falhas) são enviadas aos sites remotos para solicitar que sejam tomadas ações por lá. Embora a técnica de mensagem persistente evite o problema de bloqueio, os desenvolvedores de aplicação precisam escrever código para lidar com vários tipos de falhas.
- Os diversos esquemas de controle de concorrência utilizados em um sistema centralizado podem ser modificados para uso em um ambiente distribuído.
  - No caso de protocolos de bloqueio, a única mudança que precisa ser incorporada está no modo como o gerenciador de bloqueio é implementado. Existem diversas técnicas diferentes aqui. Um ou mais coordenadores centrais podem ser usados. Se, em vez disso, uma técnica de gerenciador de bloqueio distribuído for seguida, dados replicados precisam ser tratados de forma especial.
  - Protocolos para lidar com dados replicados incluem os protocolos de cópia primária, maioria, parcial e consenso de quórum. Estes possuem diferentes opções em termos de custo e capacidade de trabalhar na presença de falhas.
  - No caso de timestamp e esquemas de validação, a única mudança necessária é desenvolver um mecanismo para gerar timestamps globais exclusivos.
  - Muitos sistemas de banco de dados admitem a replicação lenta, em que as atualizações são propagadas para réplicas fora do escopo da transação que realizou a atualização. Essas facilidades precisam ser usadas com muito cuidado, pois podem resultar em execuções não seriáveis.
- A detecção de impasse em um ambiente de gerenciador de bloqueio distribuído requer a operação conjunta entre vários sites, pois pode haver impasses globais mesmo quando não existem impasses locais.
- Para oferecer alta disponibilidade, um banco de dados distribuído precisa detectar falhas, reconfigurar-se de modo que a computação possa continuar, e recuperar-se quando um processador ou enlace é reparado. A tarefa é

complicada ainda mais pelo fato de que é difícil distinguir entre partições de rede e falhas de site.

O protocolo da maioria pode ser estendido usando números de versão para permitir que o processamento da transação prossiga mesmo na presença de falhas. Embora o protocolo tenha uma sobrecarga significativa, ele funciona independente do tipo de falha. Protocolos menos dispendiosos estão disponíveis para lidar com as falhas no site, mas consideram que o particionamento da rede não ocorre.

- Alguns dos algoritmos distribuídos exigem o uso de um coordenador. Para oferecer alta disponibilidade, o sistema precisa manter uma cópia de backup que está pronta para assumir a responsabilidade se o coordenador falhar. Outra técnica é escolher o novo coordenador após o coordenador ter falhado. Os algoritmos que determinam qual site deverá atuar como coordenador são denominados **algoritmos de eleição**.
- As consultas sobre um banco de dados distribuído podem ter de acessar vários sites. Diversas técnicas de otimização estão disponíveis para escolher quais sites precisam ser acessados. Com base na fragmentação e replicação, essas técnicas podem usar técnicas de semijunção para reduzir a transferência de dados.
- Os bancos de dados distribuídos heterogêneos permitem que os sites tenham seus próprios esquemas e código de sistema de banco de dados. Um sistema multibanco de dados oferece um ambiente em que novas aplicações de banco de dados podem acessar dados de diversos bancos de dados preexistentes localizados em diversos ambientes de hardware e software heterogêneos. Os sistemas de banco de dados locais podem empregar diferentes modelos lógicos e linguagens de definição de dados e manipulação de dados, e podem diferir em seus mecanismos de controle de concorrência e gerenciamento de transações. Um sistema multibanco de dados cria a ilusão de integração lógica de banco de dados, sem exigir a integração física do banco de dados.
- Os sistemas de diretório podem ser vistos como uma forma especializada de banco de dados, em que as informações são organizadas em um padrão hierárquico, semelhante ao modo como os arquivos são organizados em um sistema de arquivos. Os diretórios são acessados por protocolos padronizados de acesso ao diretório, como LDAP.

Os diretórios podem ser distribuídos por vários sites, a fim de oferecer autonomia aos sites individuais. Os diretórios podem conter atestados para outros diretórios, o que ajuda a montar uma visão integrada, pela qual uma consulta é enviada a um único diretório e é executada transparentemente em todos os diretórios relevantes.

Termos de revisão

- Banco de dados distribuído homogêneo
- Banco de dados distribuído heterogêneo
- Replicação de dados
- Cópia primária
- Fragmentação de dados
  - Fragmentação horizontal
  - Fragmentação vertical
- Transparência de dados
  - Transparência de fragmentação
  - Transparência de replicação
  - Transparência de local
- Servidor de nomes
- Aliases
- Transações distribuídas
  - Transações locais
  - Transações globais
- Gerenciador de transação
- Coordenador de transação
- Modos de falha do sistema
- Partição de rede
- Protocolos commit
- Protocolo commit de duas fases (2PC)
  - Estado pronto
  - Transações em dúvida
  - Problema de bloqueio
- Protocolo commit de três fases (3PC)
- Mensagens persistentes
- Controle de concorrência
- Gerenciador de bloqueio único
- Gerenciador de bloqueio distribuído
- Protocolos para réplicas
  - Cópia primária
  - Protocolo da maioria
  - Protocolo parcial
  - Protocolo de consenso de quórum
- Timestamp
- Replicação mestre-escravo
- Replicação multimestre (atualização em qualquer lugar)
- Snapshot consistente com a transação
- Propagação lenta
- Tratamento de impasse
  - Gráfico de espera local
  - Gráfico de espera global
  - Ciclos falsos
- Disponibilidade
- Robustez
  - Técnica baseada na maioria
  - Ler em um, escrever em todos
  - ler em um, escrever em todos os disponíveis
  - Reintegração de site

- Seleção de coordenador
- Coordenador backup
- Algoritmos de eleição
- Algoritmo bully
- Processamento de consulta distribuído
- Estratégia de semijunção
- Sistema multibanco de dados
- Autonomia
- Mediadores
- Banco de dados virtual
- Sistemas de diretório
- LDAP: Lightweight Directory Access Protocol
  - Distinguished Name (DN)
  - Relative Distinguished Name (RDNs)
  - Directory Information Tree (DIT)
- Árvores de diretório distribuídas
- Sufixo DIT
- Atestado

Exercícios práticos

22.1 Como um banco de dados distribuído projetado para uma rede local poderia diferir de um projeto para uma rede remota?

22.2 Para criar um sistema distribuído altamente disponível, você precisa saber que tipos de falhas poderão ocorrer.

a. Liste os possíveis tipos de falha em um sistema distribuído.

b. Quais itens na sua lista da parte *a* também se aplicam a um sistema centralizado?

22.3 Considere uma falha que ocorre durante o 2PC para uma transação. Para cada falha possível que você listou no Exercício prático 22.2a, explique como o 2PC garante a atomicidade da transação apesar da falha.

22.4 Considere um sistema distribuído com dois sites, *A* e *B*. O site *A* pode distinguir entre os seguintes?

- *B* parado.
- O enlace entre *A* e *B* cortado.
- *B* extremamente sobrecarregado e o tempo de resposta é 100 vezes maior que o normal.

Que implicações tem a sua resposta para a recuperação nos sistemas distribuídos?

22.5 O esquema de mensagens persistentes, descrito neste capítulo, depende de timestamps combinados com o descarte de mensagens recebidas, se forem muito antigas. Sugira um esquema alternativo baseado em números de seqüência, em vez de timestamps.

22.6 Dê um exemplo em que a técnica ler um, escrever todos os disponíveis leva a um estado errôneo.

22.7 Explique a diferença entre replicação de dados em um sistema distribuído e a manutenção de um site de backup remoto.

22.8 Dê um exemplo em que a replicação lenta pode levar a um estado de banco de dados inconsistente, mesmo quando as atualizações recebem um bloqueio exclusivo sobre a cópia primária (mestre).

22.9 Considere o seguinte algoritmo de detecção de impasse. Quando a transação T_i, no site S_1, solicitar um recurso de T_j, no site S_2, uma mensagem de solicitação com timestamp n é enviada. A aresta (T_i, T_j, n) é inserida no gráfico de espera local S_1. A aresta (T_i, T_j, n) é inserida no gráfico de espera local de S_2 somente se T_j tiver recebido a mensagem de solicitação e não puder conceder imediatamente o recurso solicitado. Uma solicitação de T_i para T_j no mesmo site é tratada de modo normal; nenhum timestamp é associado à aresta (T_i, T_j). Um coordenador central invoca o algoritmo de detecção enviando uma mensagem de início a cada site no sistema.

Ao receber essa mensagem, um site envia seu gráfico de espera local ao coordenador. Observe que esse gráfico contém todas as informações locais que o site tem sobre o estado do gráfico real. O gráfico de espera reflete um estado instantâneo do site, mas não está sincronizado com relação a qualquer outro site.

Quando o controlador tiver recebido uma resposta de cada site, ele constrói um gráfico da seguinte forma:

- O grafico contém um vértice para cada transação no sistema.
- O gráfico tem uma aresta (T_i, T_j) se e somente se

Houver uma aresta (T_i, T_j) em um dos gráficos de espera.

Uma aresta (T_i, T_j, n) (para algum n) aparece em mais de um gráfico de espera.

Mostre que, se houver um ciclo no gráfico construído, então o sistema está em um estado de impasse e que, se não houver um ciclo no gráfico construído, então o sistema não estava em um estado de impasse quando a execução do algoritmo foi iniciada.

22.10 Considere uma relação que é fragmentada horizontalmente por *número_fábrica*:

funcionário(*nome, endereço, salário, número_fábrica*)

Suponha que cada fragmento tenha duas réplicas: uma armazenada no site de Nova York e uma armazenada localmente no local da fábrica. Descreva uma boa estratégia de processamento para as seguintes consultas entradas no site de San Jose.

a. Encontrar todos os funcionários na fábrica em Boca.
b. Encontrar o salário médio de todos os funcionários.
c. Encontrar o funcionário com maior salário em cada um dos seguintes sites: Toronto, Edmonton, Vancouver, Montreal.
d. Encontrar o funcionário com menor salário na empresa.

22.11 Calcular $r \times s$ para as relações da Figura 22.7.

22.12 Dado que a funcionalidade LDAP pode ser implementada em cima de um sistema de banco de dados, qual é a necessidade para o padrão LDAP?

Exercícios

22.13 Discuta as vantagens relativas dos bancos de dados centralizados e distribuídos.

22.14 Explique a diferença entre os seguintes: transparência da fragmentação, transparência da replicação e transparência de local.

22.15 Quando é útil ter replicação ou fragmentação de dados? Explique a sua resposta.

22.16 Explique as noções de transparência e autonomia. Por que essas noções são desejáveis de um ponto de vista de fatores humanos?

22.17 Se aplicarmos uma versão distribuída do protocolo de granularidade múltipla do Capítulo 16 a um banco de dados distribuído, o site responsável pela raiz do DAG pode se tornar um gargalo. Suponha que modifiquemos esse protocolo da seguinte maneira:

- Somente os bloqueios do modo de intenção são permitidos na raiz.
- Todas as transações recebem todos os bloqueios do modo de intenção possíveis sobre a raiz automaticamente.

Mostre que essas modificações aliviam aquele problema sem permitir quaisquer escalonamentos não seriáveis.

22.18 Estude e resuma as facilidades que o sistema de banco de dados que você está usando oferece para lidar com estados inconsistentes que possam ser alcançados com a propagação lenta de atualizações.

22.19 Discuta as vantagens e as desvantagens dos dois métodos que apresentamos na seção "Timestamp" para gerar timestamps globalmente exclusivos.

22.20 Considere as relações

funcionário (nome, endereço, salário, número_fábrica)
máquina (número_máquina, tipo, número_fábrica)

Considere que a relação está fragmentada horizontalmente por *número_fábrica*, e que cada fragmento é armazenado localmente no site de sua fábrica correspondente. Suponha que a relação *máquina* esteja armazenada inteiramente no site de Armonk. Descreva uma boa estratégia para processar cada uma das seguintes consultas.

a. Encontrar todos os funcionários na fábrica que contêm o número 1130.
b. Encontrar todos os funcionários nas fábricas que contêm máquinas cujo tipo é "moenda".
c. Encontrar todas as máquinas na fábrica em Almaden.
d. Encontrar funcionário $\bowtie$ máquina.

22.21 Para cada uma das estratégias do Exercício 22.20, indique como a sua escolha de uma estratégia depende:

a. Do site em que a consulta foi inserida
b. Do site em que o resultado é desejado

22.22 A expressão $r_i \times r_j$ é necessariamente igual a $r_j \times r_i$? Sob que condições $r_i \times r_j = r_j \times r_i$ se mantém?

22.23 Descreva como o LDAP pode ser usado para oferecer múltiplas visões hierárquicas dos dados, sem replicar os dados em nível básico.

Notas bibliográficas

Livros-texto sobre bancos de dados distribuídos são oferecidos por Ozsu e Valduriez [1999] e Ceri e Pelagatti [1984]. Redes de computadores são discutidas em Tanenbaum [2002] e Halsall [1996]. Breitbart *et al.* [1999b] apresentam uma visão geral dos bancos de dados distribuídos.

A	*B*	*C*
1	2	3
4	5	6
1	2	4
5	3	2
8	9	7

r

C	*D*	*E*
3	4	5
3	6	8
2	3	2
1	4	1
1	2	3

s

Figura 22.7 Relações para o Exercício prático 22.11.

A implementação do conceito de transação em um banco de dados distribuído é apresentada por Gray [1981], Traiger *et al.* [1982], Spector e Schwarz [1983] e Eppinger *et al.* [1991]. O protocolo 2PC foi desenvolvido por Lampson e Sturgis [1976] e Gray [1978]. O protocolo commit de três fases é de Skeen [1981]. Mohan e Lindsay [1983] discutem duas versões modificadas do 2PC, chamadas *presume commit* e *presume abort*, que reduzem a sobrecarga do 2PC, definindo suposições-padrão com relação ao destino das transações.

O algoritmo bully da seção "Seleção do coordenador" é de Garcia-Molina [1982]. O sincronismo de clock distribuído é discutido em Lamport [1978]. O controle de concorrência distribuído é abordado por Menasce *et al.* [1980], Bernstein e Goodman [1980], Bernstein e Goodman [1981] e Bernstein e Goodman [1982].

O gerenciador de transações do R* é descrito em Mohan *et al.* [1986]. O controle de concorrência para dados replicados, baseado no conceito de votação, é apresentado por Gifford [1979] e Thomas [1979]. As técnicas de validação para os esquemas de controle de concorrência distribuídos são descritas por Schlageter [1981], Ceri e Owicki [1983] e Bassiouni [1988].

O problema das atualizações concorrentes aos dados replicados foi revisado no contexto dos depósitos de dados por Gray *et al.* [1996]. Anderson *et al.* [1998] discutem questões referentes à replicação lenta e consistência. Breitbart *et al.* [1999a] descrevem os protocolos de atualização lentos para o tratamento da replicação.

Os manuais de usuário de diversos sistemas de banco de dados oferecem detalhes de como eles tratam a replicação e consistência. Huang e Garcia-Molina [2001] enfatizam a semântica "exatamente uma vez" em um sistema de mensagens replicado.

Knapp [1987] faz um levantamento da literatura de detecção de impasse distribuída. O Exercício prático 22.9 é de Stuart *et al.* [1984].

O processamento de consulta distribuído é discutido em Wong [1977], Epstein *et al.* [1978], Hevner e Yao [1979] e Epstein e Stonebraker [1980].

Selinger e Adiba [1980] e Daniels *et al.* [1982] discutem a técnica de processamento de consulta distribuído tomada pelo R* (uma versão distribuída do System R). Mackert e Lohman [1986] oferecem uma avaliação de desempenho dos algoritmos de processamento de consulta no R*.

A otimização de consulta dinâmica nos multibancos de dados é tratada por Ozcan e outros [1997]. Adali *et al.* [1996] e Papakonstantinou *et al.* [1996] descrevem as questões de otimização de consulta em sistemas mediadores.

Weltman e Dahbura [2000] e Howes *et al.* [1999] oferecem um livro-texto sobre LDAP. Kapitskaia *et al.* [2000] descrevem aspectos de caching de dados de diretório LDAP.

PARTE 8

Outros tópicos

O Capítulo 23 aborda uma série de questões no ajuste de desempenho do sistema de banco de dados para melhorar a velocidade da aplicação. Ele também discute benchmarks que são usados como medidas de desempenho de sistemas de banco de dados comercial. Depois discutimos o processo de padronização e os padrões existentes da linguagem de banco de dados. Ele conclui com uma discussão do papel dos sistemas de banco de dados no comércio eletrônico, dos desafios da manutenção de acesso aos dados armazenados nos "sistemas legados" mais antigos e da migração de uma aplicação para um novo sistema.

O Capítulo 24 descreve os tipos de dados, como dados temporais, dados espaciais e dados de multimídia, e as questões no armazenamento desses dados nos bancos de dados. Aplicações como computação móvel e suas conexões com bancos de dados, também são descritas neste capítulo.

Finalmente, o Capítulo 25 descreve diversas técnicas avançadas de processamento de transação, incluindo monitores de processamento de transação, fluxos de trabalho transacionais, transações de longa duração e transações multibanco de dados.

CAPÍTULO 23

Desenvolvimento avançado de aplicações

Existem diversas tarefas no desenvolvimento de aplicações. Já vimos, nos Capítulos 6 a 8, como projetar e montar uma aplicação. Um dos aspectos do projeto de aplicações é o desempenho que alguém espera da aplicação. De fato, é comum descobrir que, quando uma aplicação é montada, ela fica mais lenta do que os projetistas desejavam, ou trata de menos transações por segundo do que eles precisavam. Uma aplicação que leva um tempo excessivo para realizar as ações solicitadas pode causar, pelo menos, insatisfação do usuário e, no pior dos casos, pode se tornar completamente inutilizável.

As aplicações podem ser criadas de modo a serem executadas muito mais rapidamente com o ajuste do desempenho, que consiste em localizar e eliminar gargalos e acrescentar hardware apropriado, como memória ou discos. Existem muitas ações que um desenvolvedor de aplicações pode fazer para ajustar a aplicação, e existem ações que um administrador de sistemas de banco de dados pode fazer para agilizar o processamento para uma aplicação.

Os benchmarks são conjuntos padronizados de tarefas que ajudam a caracterizar o desempenho dos sistemas de banco de dados. Eles são úteis para dar uma idéia aproximada dos requisitos de hardware e software de uma aplicação, antes mesmo que a aplicação seja criada.

Os padrões são muito importantes para o desenvolvimento de aplicações, especialmente na era da Internet, pois as aplicações precisam se comunicar entre si para realizar tarefas úteis. Diversos padrões foram propostos, afetando o desenvolvimento de aplicações de banco de dados.

Os sistemas legados são sistemas de aplicações que estão desatualizados e normalmente são baseados em tecnologia de uma geração mais antiga. Porém, eles normalmente estão no núcleo das organizações, e executam aplicações de missão crítica. Explicamos as questões de interface e os problemas na migração dos sistemas legados, substituindo-os por sistemas mais modernos.

Ajuste de desempenho

O ajuste de desempenho de um sistema envolve ajustar vários parâmetros e opções de projeto a fim de melhorar seu desempenho para uma aplicação específica. Diversos aspectos de um projeto de sistema de banco de dados – variando desde aspectos de alto nível, como o projeto de esquema e transação, até parâmetros de banco de dados, como tamanhos de buffer, e aspectos de hardware, como número de discos – afetam o desempenho de uma aplicação. Cada um desses aspectos pode ser ajustado de modo que o desempenho seja melhorado.

Localização dos gargalos

O desempenho da maioria dos sistemas (pelo menos antes de serem ajustados) em geral é limitado principalmente pelo desempenho de um ou alguns componentes, chamados gargalos. Por exemplo, um programa pode gastar 80% do seu tempo em um pequeno loop dentro do código e 20% no restante do código; o pequeno loop, então, é um gargalo. A melhoria do desempenho de um componente que não é um gargalo faz pouco para melhorar a velocidade geral do sistema; no exemplo, a melhoria da velocidade do restante do código não pode levar a mais do que 20% de melhoria geral, enquanto a melhoria da velocidade do loop de gargalo poderia resultar em uma melhoria de quase 80% em geral, na melhor das hipóteses.

Logo, ao ajustar um sistema, primeiro é preciso tentar descobrir quais são os gargalos e depois eliminá-los melhorando o desempenho dos componentes do sistema que

os causam. Quando um gargalo é removido, pode acontecer que outro componente se torne o gargalo. Em um sistema bem balanceado, nenhum componente isolado é o gargalo. Se o sistema tiver gargalos, os componentes que não fazem parte do gargalo são subutilizados e talvez possam ser substituídos por componentes mais baratos, com menor desempenho.

Para programas simples, o tempo gasto em cada região do código determina o tempo de execução geral. Porém, os sistemas de banco de dados são muito mais complexos e podem ser modelados como **sistemas de enfileiramento**. Uma transação solicita diversos serviços do sistema de banco de dados, começando desde a entrada em um processo servidor, leituras de disco durante a execução, ciclos de CPU e bloqueios para controle de concorrência. Cada um desses serviços possui uma fila associada a ele, e pequenas transações podem gastar a maior parte do seu tempo esperando em filas – especialmente em filas de E/S de disco – em vez de executar código. A Figura 23.1 ilustra algumas das filas em um sistema de banco de dados.

Como resultado das diversas filas no banco de dados, gargalos em um sistema de banco de dados normalmente aparecem na forma de filas longas para um serviço em particular ou, de forma equivalente, em altas utilizações para um serviço em particular. Se as solicitações forem espaçadas de modo exatamente uniforme, e o tempo para atender uma solicitação for menor ou igual ao tempo antes que a próxima solicitação chegue, então cada solicitação achará o recurso ocioso e, portanto, pode iniciar a execução logo, sem esperar. Infelizmente, a chegada de solicitações em um sistema de banco de dados nunca é tão uniforme, e sim aleatória.

Se um recurso, como um disco, tiver uma utilização baixa, então, quando uma solicitação for feita, o recurso provavelmente estará ocioso, quando o tempo de espera para a solicitação será 0. Supondo chegadas uniforme e aleatoriamente distribuídas, o tamanho da fila aumenta rapidamente, resultando em tempos de espera excessivamente longos. A utilização de um recurso deverá ser mantida baixa o suficiente para que o tamanho da fila seja curto. Via de regra, as utilizações em torno de 70% são consideradas como boas, e as utilizações acima de 90% são consideradas excessivas, pois resultarão em atrasos significativos. Para descobrir mais sobre a teoria dos sistemas de enfileiramento, geralmente chamada de **teoria do enfileiramento**, você pode consultar as referências citadas nas notas bibliográficas.

Parâmetros ajustáveis

Os administradores de banco de dados podem ajustar o sistema de banco de dados em três níveis. O nível mais baixo está no nível de hardware. As opções para ajustar sistemas nesse nível incluem acrescentar discos ou usar um sistema RAID se a E/S de disco for um gargalo, aumentar a quantidade de memória se o tamanho do buffer de disco for um gargalo ou passar para um processador mais rápido se o uso de CPU for um gargalo.

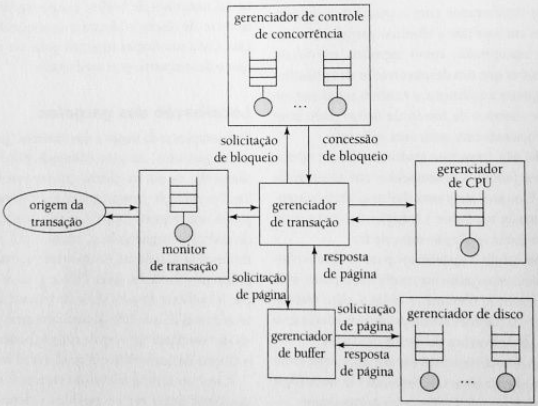


Figura 23.1 Filas em um sistema de banco de dados.

O segundo nível consiste nos parâmetros do sistema de banco de dados, como tamanho de buffer ou intervalos de pontos de verificação. O conjunto exato de parâmetros do sistema de banco de dados que podem ser ajustados depende do sistema de banco de dados específico. A maioria dos manuais de sistema de banco de dados oferece informações sobre quais parâmetros do sistema de banco de dados podem ser ajustados e como você deverá escolher valores para eles. Sistemas de banco de dados bem projetados realizam o máximo de desempenho possível automaticamente, liberando o usuário ou o administrador de banco de dados do peso. Por exemplo, em muitos sistemas de banco de dados, o tamanho do buffer é fixo, mas ajustável. Se o sistema ajusta automaticamente o tamanho do buffer observando indicadores como taxas de falta de página, então o usuário não terá de se preocupar com o ajuste do tamanho do buffer.

O terceiro nível é o nível mais alto. Ele inclui o esquema e as transações. O administrador pode ajustar o projeto do esquema, os índices que são criados e as transações que são executadas, para melhorar o desempenho. O ajuste nesse nível é comparativamente independente do sistema.

Os três níveis de ajuste interagem entre si; temos de considerá-los juntos ao ajustar um sistema. Por exemplo, o ajuste em um nível mais alto pode resultar no gargalo de hardware passando do sistema de disco para a CPU, ou vice-versa.

Ajuste do hardware

Até mesmo em um sistema de processamento de transação bem projetado, cada transação normalmente precisa realizar pelo menos algumas operações de E/S, se os dados exigidos pela transação estiverem em disco. Um fator importante no ajuste de um sistema de processamento de transação é certificar-se de que o subsistema de disco pode lidar com a taxa em que as operações de E/S são exigidas. Por exemplo, considere um disco que admita a um tempo de acesso de cerca de 10 milissegundos, e uma taxa de transferência média de 25 megabytes por segundo (um disco muito comum hoje). Esse disco admitiria um pouco menos de 100 operações de E/S de acesso aleatório de 4 kilobytes cada por segundo. Se cada transação exigir apenas 2 operações de E/S, um único disco admitiria no máximo 50 transações por segundo. A única maneira de admitir mais transações por segundo é aumentar o número de discos. Se o sistema precisar admitir n transações por segundo, cada uma realizando 2 operações de E/S, os dados precisam ser divididos (ou particionados de alguma forma) por pelo menos $n/50$ discos (ignorando a distorção).

Observe aqui que o fator limitador não é a capacidade do disco, mas a velocidade em que os dados aleatórios podem ser acessados (limitada, por sua vez, pela velocidade em que o braço do disco pode mover). O número de operações de E/S por transação pode ser reduzido armazenando-se mais dados na memória. Se todos os dados estiverem na memória, não haverá E/S de disco, exceto para escritas. Manter dados usados com freqüência na memória reduz o número de E/Ss de disco, e compensa o custo extra de memória. Manter dados usados com pouca freqüência na memória seria um desperdício, pois a memória é muito mais cara que o disco.

A questão é, para determinada quantidade de dinheiro disponível para gastar em discos ou memória, qual é a melhor maneira de gastar o dinheiro de modo a conseguir o número máximo de transações por segundo? Uma redução de 1 E/S por segundo economiza

(preço por unidade de disco)/(acesso por segundo por disco)

Assim, se determinada página for acessada n vezes por segundo, a economia necessária para mantê-la na memória é n vezes esse valor. Armazenar uma página na memória custa

(preço por megabyte de memória)/
(páginas por megabyte de memória)

Assim, o ponto de equilíbrio é

$$n * \frac{\text{preço por unidade de disco}}{\text{acesos por segundo por disco}} = \frac{\text{preço por megabyte de memória}}{\text{páginas por megabyte de memória}}$$

Podemos modificar a equação e substituir os valores atuais para cada um desses parâmetros a fim de chegar a um valor para n; se uma página é acessada com mais freqüência do que isso, vale a pena comprar memória suficiente para armazená-la. A tecnologia de disco atual e os preços de memória e disco dão um valor de n em torno de 1/300 vezes por segundo (ou, de forma equivalente, uma vez em 5 minutos) para páginas acessadas aleatoriamente.

Esse raciocínio é capturado pela regra prática chamada **regra dos 5 minutos**: se uma página é usada com mais freqüência do que uma vez em 5 minutos, ela deve ser colocada em cache na memória. Em outras palavras, vale a pena comprar memória suficiente para colocar em cache todas as páginas que são acessadas pelo menos uma vez em 5 minutos na média. Para os dados que são acessados com menos freqüência, compre discos suficientes para admitir a taxa de E/S exigida para os dados.

A fórmula para encontrar o ponto de equilíbrio depende de fatores, como os custos de discos e de memória, que foram alterados por fatores de 100 ou 1.000 na última década. Porém, é interessante observar que as razões das mudanças foram tais que o ponto de equilíbrio permaneceu em aproximadamente 5 minutos; a regra dos 5 minutos não mudou para, digamos, regra da 1 hora ou regra do 1 segundo!

Para os dados que são acessados seqüencialmente, muito mais páginas podem ser lidas por segundo. Considerando que 1 megabyte de dados são lidos de cada vez, obtemos a **regra do 1 minuto**, que diz que os dados acessados seqüencialmente devem ser mantidos em cache na memória se forem usados pelo menos uma vez em 1 minuto.

As regras práticas levam em conta somente o número de operações de E/S, e não consideram fatores como o tempo de resposta. Algumas aplicações precisam manter na memória até mesmo dados usados com pouca freqüência, para dar suporte a tempos de resposta que são menores do que ou comparáveis ao tempo de acesso do disco.

Outro aspecto do ajuste é se será usado RAID 1 ou RAID 5. A resposta depende da freqüência com que os dados são atualizados, pois RAID 5 é muito mais lento do que RAID 1 nas escritas aleatórias: RAID 5 exige 2 leituras e 2 escritas para executar uma única solicitação de escrita aleatória. Se uma aplicação realiza r leituras aleatórias e w escritas aleatórias por segundo para dar suporte a determinado throughput, uma implementação RAID 5 exigiria $r + 4w$ operações de E/S por segundo, enquanto uma implementação RAID 1 exigiria $r + w$ operações de E/S por segundo. Podemos, então, calcular o número de discos necessários para dar suporte às operações de E/S exigidas por segundo dividindo o resultado do cálculo por 100 operações de E/S por segundo (para os discos da geração atual). Para muitas aplicações, r e w são tão grandes que os $(r+w)/100$ discos podem facilmente manter duas cópias de todos os dados. Para tais aplicações, se RAID 1 for usado, o número exigido de discos é, na realidade, menor que o número exigido de discos se RAID 5 for usado! Assim, RAID 5 é útil somente quando os requisitos de armazenamento de dados são muito grandes, mas as taxas de E/S e os requisitos de transferência de dados são pequenos, ou seja, para dados muito grandes e muito "frios".

Ajustando o esquema

Dentro das restrições da forma normal escolhida, é possível particionar relações verticalmente. Por exemplo, considere a relação *conta*, com o esquema

conta(número_conta, nome_agência, saldo)

para o qual *número_conta é uma chave. Dentro das restrições das formas normais (BCNF e terceira forma normal), podemos particionar a relação conta* em duas relações:

agência_conta (número_conta, nome_agência)
saldo_conta (número_conta, saldo)

As duas representações são logicamente equivalentes, pois *número_conta* é uma chave, mas elas possuem características de desempenho diferentes.

Se a maioria dos acessos a informações de conta examinarem apenas *número_conta* e *saldo*, então eles podem ser executados sobre a relação *saldo_conta*, e o acesso provavelmente será um pouco mais rápido, pois o atributo *nome_agência* não é apanhado. Pelo mesmo motivo, mais tuplas de *saldo_conta* caberão no buffer do que as tuplas correspondentes de *conta*, novamente levando a um desempenho mais rápido. Esse efeito seria particularmente marcado se o atributo *nome_agência* fosse grande. Logo, um esquema consistindo em *agência_conta* e *saldo_conta* seria preferível a um esquema consistindo na relação *conta* nesse caso.

Por outro lado, se a maioria dos acessos a informações de conta exigir *saldo* e *nome_agência*, o uso da relação *conta* seria preferível, pois o custo da junção de *saldo_conta* e *agência_conta* seria evitado. Além disso, a sobrecarga de armazenamento seria menor, pois haveria apenas uma relação, e o atributo *número_conta* não seria replicado.

Outro truque para melhorar o desempenho é armazenar uma **relação desnormalizada**, como uma junção de *conta* e *depositante*, em que a informação sobre nomes de agência e saldos é repetida para cada cliente de conta. Mais esforço terá de ser feito para garantir que a relação estará consistente sempre que uma atualização for executada. Porém, uma consulta que obtém os nomes dos clientes e os saldos associados será agilizada, pois a junção de *conta* e *depositante* terá sido previamente calculada. Se tal consulta for executada com freqüência e tiver de ser realizada da forma mais eficiente possível, a relação desnormalizada poderia ser benéfica.

As visões materializadas oferecem os benefícios que as relações desnormalizadas oferecem, ao custo de algum armazenamento extra; descrevemos o ajuste do desempenho das visões materializadas na seção "Usando visões materializadas". Uma grande vantagem das visões materializadas em cima das relações desnormalizadas é que a manutenção da consistência dos dados redundantes se torna tarefa do sistema de banco de dados, e não do programador. Assim, as visões materializadas são preferíveis, sempre que foram aceitas pelo sistema de banco de dados.

Outra técnica para agilizar a computação da junção sem materializá-la é agrupar registros que combinariam na junção sobre a mesma página de disco. Vimos essas organizações de arquivo agrupadas na seção "Organização de arquivos com agrupamento de múltiplas tabelas" do Capítulo 11.

Ajuste de índices

Podemos ajustar os índices em um sistema de banco de dados para melhorar o desempenho. Se as consultas forem o gargalo, normalmente poderemos agilizá-las criando índices apropriados sobre relações. Se as atualizações forem o

gargalo, pode haver muitos índices, que precisam ser atualizados quando as relações forem atualizadas. A remoção de índices pode agilizar certas atualizações.

A escolha do tipo de índice também é importante. Alguns sistemas de banco de dados admitem diferentes tipos de índices, como índices de hash e índices de árvore B. Se as consultas de intervalo forem comuns, os índices de árvore B são preferíveis aos índices de hash. Se um índice deve se tornar um índice agrupado, isso é outro parâmetro ajustável. Somente um índice sobre uma relação pode se tornar agrupado, armazenando a relação classificada sobre os atributos de índice. Em geral, o índice que beneficia o maior número de consultas e atualizações deve ser agrupado.

Para ajudar a identificar quais índices devem ser criados e qual índice (se houver) sobre cada relação deve ser agrupado, a maioria dos sistemas de banco de dados comerciais oferece *assistentes de ajuste*; estes são descritos com mais detalhes na seção "Ajuste automatizado do projeto físico". Essas ferramentas usam o histórico passado das consultas e atualizações (chamado *carga de trabalho*) para estimar os efeitos de diversos índices sobre o tempo de execução das consultas e atualizações na carga de trabalho. As recomendações sobre quais índices criar são baseadas nessas estimativas.

Usando visões materializadas

A manutenção de visões materializadas pode agilizar bastante certos tipos de consultas, em particular, consultas de agregação. Lembre-se do exemplo da seção "Views materializadas" do Capítulo 14, no qual a quantidade total de empréstimos em cada agência (obtida pela soma das quantias de empréstimo de todos os empréstimos na agência) é exigida com freqüência. Como vimos naquela seção, a criação de uma visão materializada armazenando a quantia total do empréstimo para cada agência pode agilizar bastante tais consultas.

Contudo, as visões materializadas devem ser usadas com cuidado, pois não existe apenas uma sobrecarga de espaço para armazená-las, mas, ainda mais importante, há também uma sobrecarga de tempo para manter visões materializadas. No caso de **manutenção de visão imediata**, se as atualizações de uma transação afetarem a visão materializada, esta precisa ser atualizada como parte da mesma transação. A transação pode, portanto, executar mais lentamente. No caso da **manutenção de visão adiada**, a visão materializada é atualizada depois; até que seja atualizada, a visão materializada pode ser inconsistente com as relações do banco de dados. Por exemplo, a visão materializada pode ser atualizada quando uma consulta usar a visão, ou periodicamente. O uso da manutenção adiada reduz o peso sobre transações de atualização.

Uma questão importante é: como alguém seleciona quais visões materializadas manter? O administrador do sistema pode fazer a seleção manualmente, examinando os tipos de consultas na carga de trabalho e descobrindo quais consultas precisam ser executadas mais rapidamente e quais atualizações/consultas podem ser executadas mais lentamente. Examinando, o administrador do sistema pode escolher um conjunto apropriado de visões materializadas. Por exemplo, o administrador poderá descobrir que certa agregação é usada com freqüência e decidir materializá-la, ou poderá descobrir que determinada junção é calculada com freqüência e decidir materializá-la.

Contudo, a escolha manual é cansativa até mesmo para conjuntos moderadamente grandes de tipos de consulta, e pode ser difícil fazer uma boa escolha, pois isso exige compreender os custos das diferentes alternativas; somente o otimizador da consulta pode estimar os custos com precisão razoável, sem realmente executar a consulta. Assim, um bom conjunto de visões pode ser encontrado apenas por tentativa e erro – ou seja, materializando uma ou mais visões, executando a carga de trabalho e medindo o tempo gasto para executar as consultas na carga de trabalho. O administrador repete o processo até que um conjunto de visões seja encontrado e ofereça desempenho aceitável.

Uma alternativa melhor é oferecer suporte para selecionar visões materializadas dentro do próprio sistema de banco de dados, integrado ao otimizador de consulta. Essa técnica é descrita com mais detalhes na próxima seção.

Ajuste automatizado do projeto físico

A maioria dos sistemas de banco de dados comerciais de hoje oferece ferramentas para ajudar o administrador de banco de dados com a seleção de índice e visão materializada e outras tarefas relacionadas ao projeto físico do banco de dados, por exemplo, como particionar dados em um sistema de banco de dados paralelo.

Essas ferramentas examinam a carga de trabalho (o histórico das consultas e atualizações) e sugerem índices e visões a serem materializadas. O usuário da ferramenta de ajuste pode especificar a importância de agilizar diferentes consultas, que o usuário leva em conta quando seleciona visões para materializar. Normalmente, o ajuste precisa ser feito antes que a aplicação seja totalmente desenvolvida, e o conteúdo real do banco de dados pode ser pequeno no banco de dados de desenvolvimento, mas espera-se que seja muito maior em um banco de dados de produção. Assim, algumas ferramentas de ajuste também permitem que o usuário da ferramenta de ajuste especifique informações sobre o tamanho esperado do banco de dados e estatísticas relacionadas.

O Database Tuning Assistant da Microsoft, por exemplo, permite que o usuário faça perguntas hipotéticas, pelas quais o usuário pode selecionar uma visão, e o otimizador então

estima o efeito de materializar a visão sobre o custo total da carga de trabalho e sobre os custos individuais dos diferentes tipos de consulta/atualização na carga de trabalho.

As técnicas de seleção de índice automatizado e visão materializada normalmente são implementadas pela enumeração de diferentes alternativas e pelo uso do otimizador de consulta para estimar os custos e os benefícios da seleção de cada alternativa usando a carga de trabalho. Como o número de alternativas de projeto pode ser extremamente grande, como também a carga de trabalho, as técnicas de seleção precisam ser cuidadosamente projetadas.

O primeiro passo é gerar uma carga de trabalho. Isso normalmente é feito registrando-se todas as consultas e atualizações que são executadas durante algum período de tempo. Em seguida, as ferramentas de seleção realizam *compactação de carga de trabalho*, ou seja, criam uma representação da carga de trabalho usando um pequeno número de atualizações e consultas. Por exemplo, as atualizações da mesma forma podem ser representadas por uma única atualização com o peso correspondente a quantas vezes a atualização ocorreu. As consultas da mesma forma podem ser substituídas de modo semelhante por uma representativa com peso apropriado. Depois disso, as consultas que são muito pouco freqüentes e não possuem um custo alto podem ser descartadas de consideração. As consultas mais dispendiosas podem ser escolhidas para serem resolvidas primeiro. Essa compactação de carga de trabalho é essencial para grandes cargas de trabalho.

Com a ajuda do otimizador, a ferramenta chegaria a um conjunto de índices e visões materializadas que poderia ajudar as consultas e atualizações na carga de trabalho compactada. Diferentes combinações desses índices e visões materializadas podem ser experimentadas para se encontrar a melhor. Porém, uma técnica completa seria totalmente impraticável, pois o número de índices e visões materializadas em potencial já é grande, e cada subconjunto destas é uma alternativa de projeto em potencial, levando a um número exponencial de alternativas. Heurísticas são usadas para reduzir o espaço das alternativas, ou seja, para reduzir o número de combinações consideradas.

Heurísticas gulosas para seleção de índice e visão materializada operam da seguinte forma: elas estimam os benefícios da materialização de diferentes índices/visões (usando a funcionalidade de estimativa de custo do otimizador como sub-rotina). Elas então escolhem o índice/visão que oferece o máximo de benefício ou o máximo de benefício por espaço unitário (ou seja, benefício dividido pelo espaço exigido para armazenar o índice ou visão). O custo de manter o índice/visão precisa ser levado em conta no cálculo do benefício. Quando a heurística tiver selecionado um índice/visão, os benefícios de outros índices/visões podem ter mudado, de modo que a heurística os recalcula e escolhe o próximo índice/visão para materialização. O processo continua até que o espaço de disco disponível para armazenar índices ou as visões materializadas sejam esgotados, ou o custo de manter os candidatos restantes seja maior que o benefício para as consultas que poderiam usar os índices/visões.

As ferramentas de seleção de índice e visão materializada do mundo real normalmente incorporam alguns elementos da seleção gulosa, mas utilizam outras técnicas para obter melhores resultados. Elas também admitem outros aspectos do projeto físico do banco de dados, como a decisão de como particionar uma relação em um banco de dados paralelo, ou que mecanismo de armazenamento físico deve ser usado para uma relação.

Ajuste de transações

Nesta seção, estudamos duas técnicas para melhorar o desempenho da transação:

- Melhorar a orientação do conjunto
- Reduzir a disputa pelo bloqueio

No passado, os otimizadores em muitos sistemas de banco de dados não eram particularmente bons, e o modo como uma consulta era escrita teria uma grande influência no modo como era executada e, portanto, sobre o desempenho. Os otimizadores avançados de hoje podem transformar até mesmo consultas mal escritas e executá-las de modo eficiente, de modo que a necessidade de ajustar consultas individuais é menos importante do que antes. Porém, consultas complexas contendo subconsultas aninhadas não são otimizadas muito bem por muitos otimizadores. A maioria dos sistemas oferece um mecanismo para descobrir o plano de execução exato para uma consulta; essa informação pode ser usada para reescrever a consulta em uma forma que o otimizador possa trabalhar melhor.

Na SQL embutida, se uma consulta for executada com freqüência com diferentes valores para um parâmetro, pode ser importante combinar chamadas para uma consulta mais orientada a conjunto, que é executada apenas uma vez. Em geral, o custo de comunicação das consultas SQL é relativamente alto nos sistemas cliente-servidor, de modo que a combinação das chamadas SQL embutidas é particularmente útil em tais sistemas.

Por exemplo, considere um programa que percorre cada departamento especificado em uma lista, invocando uma consulta SQL embutida para encontrar as despesas totais do departamento, usando a construção **group by** sobre uma relação *despesas*(*data*, *funcionário*, *departamento*, *quantia*). Se a relação *despesas* não tiver um índice agrupado sobre *departamento*, cada consulta desse tipo resultará

em uma varredura da relação. Em vez disso, podemos usar uma única consulta SQL para encontrar as despesas totais de todos os departamentos; a consulta pode ser avaliada com uma única varredura. Os departamentos relevantes podem então ser examinados nessa relação temporária (muito menor) contendo a agregação. Mesmo que exista um índice que permita o acesso eficiente às tuplas de determinado departamento, o uso de múltiplas consultas SQL pode ter uma alta sobrecarga de comunicação em um sistema cliente-servidor. O custo da comunicação pode ser reduzido pelo uso de uma única consulta SQL, levando seus resultados para o lado do cliente, e depois percorrendo os resultados para encontrar as tuplas necessárias.

Outra técnica muito utilizada nos sistemas cliente-servidor para reduzir o custo da comunicação e da compilação SQL é usar procedimentos armazenados, em que as consultas são armazenadas no servidor na forma de procedimentos, que podem ser pré-compilados. Os clientes podem invocar esses procedimentos armazenados, em vez de comunicar consultas inteiras.

A execução simultânea de diferentes tipos de transações às vezes pode levar a um desempenho fraco, devido à disputa pelos bloqueios. Considere, por exemplo, um banco de dados bancário. Durante o dia, diversas pequenas transações de atualização são executadas quase continuamente. Suponha que uma consulta grande, que calcula estatísticas sobre agências, seja executada ao mesmo tempo. Se a consulta realizar uma varredura sobre uma relação, ela pode bloquear todas as atualizações sobre a relação enquanto é executada, e isso pode ter um efeito desastroso sobre o desempenho do sistema.

Alguns sistemas de banco de dados – Oracle e Microsoft SQLServer, por exemplo – permitem o controle de concorrência multiversão, por onde as consultas são executadas sobre um snapshot dos dados, e as atualizações podem continuar simultaneamente. Esse recurso deverá ser usado sempre que estiver disponível. Se não estiver disponível, uma opção alternativa é executar consultas grandes em momentos em que as atualizações são poucas ou inexistentes. Para bancos de dados que dão suporte a sites, pode não haver um período de silêncio para as atualizações.

Outra alternativa é usar níveis de consistência mais fracos, pelos quais a avaliação da consulta possui um impacto mínimo sobre as atualizações concorrentes, mas os resultados da consulta não têm garantias de serem consistentes. A semântica da aplicação determina se respostas aproximadas (inconsistentes) são aceitáveis. As aplicações normalmente mantêm contadores de número de seqüência atualizados por muitas transações, que podem se tornar pontos de disputa por bloqueio. O Exercício prático 23.1 explora como os contadores de seqüência fornecidos pelo banco de dados podem ajudar a melhorar a concorrência, usando níveis de consistência mais fracos.

Transações de atualização longas podem causar problemas de desempenho com logs do sistema e aumentar o tempo gasto para se recuperar de falhas do sistema. Se uma transação realizar muitas atualizações, o log do sistema pode se tornar cheio, mesmo antes de a transação terminar, quando ela terá de ser revertida. Se uma transação de atualização for executada por um longo período de tempo (mesmo com poucas atualizações), ela pode bloquear a exclusão de antigas partes do log, se o sistema de logging não for bem projetado. Novamente, esse bloqueio poderia levar a um log repleto.

Para evitar esses problemas, muitos sistemas de banco de dados impõem limites estritos sobre o número de atualizações que uma única transação pode executar. Mesmo que o sistema não imponha tais limites, normalmente é útil dividir uma transação de atualização grande em um conjunto de transações de atualização menores, onde for possível. Por exemplo, uma transação que dá um aumento a cada funcionário de uma grande empresa poderia ser dividida em uma série de transações pequenas, cada qual atualizando um pequeno intervalo de matrículas de funcionário. Essas transações são denominadas **transações de mini-batch**. Porém, elas precisam ser usadas com cautela. Primeiro, se houver atualizações simultâneas no conjunto de funcionários, o resultado do conjunto de transações menores pode não ser equivalente ao da única transação grande. Segundo, se houver uma falha, os salários de alguns dos funcionários teriam sido aumentados pelas transações confirmadas, mas os salários de outros funcionários não seriam. Para evitar esse problema, assim que o sistema se recuperar da falha, temos de executar as transações restantes no batch.

Simulação de desempenho

Para testar o desempenho de um sistema de banco de dados mesmo antes que seja instalado, podemos criar um modelo de simulação de desempenho do sistema de banco de dados. Cada serviço mostrado na Figura 23.1, como CPU, disco, buffer e controle de concorrência, é modelado na simulação. Em vez de modelar detalhes de um serviço, o modelo de simulação só pode capturar alguns aspectos de cada serviço, como a **tempo de serviço** – ou seja, o tempo gasto para terminar de processar uma solicitação uma vez que o processamento tenha se iniciado. Assim, a simulação pode modelar um acesso ao disco a partir apenas do tempo médio de acesso ao disco.

Como as solicitações para um serviço geralmente precisam esperar sua vez, cada serviço possui uma fila associada no modelo de simulação. Uma transação consiste em uma

série de solicitações. As solicitações são enfileiradas à medida que chegam, e são atendidas de acordo com a política para esse serviço, como "primeiro a chegar, primeiro a ser atendido". Os modelos para serviços como CPU e discos conceitualmente operam em paralelo, para levar em conta o fato de que esses subsistemas operam em paralelo em um sistema real.

Quando o modelo de simulação para processamento de transação é criado, o administrador do sistema pode executar diversas experiências sobre ele. O administrador pode usar experiências com transações simuladas chegando em diferentes taxas para descobrir como o sistema se comportaria sob diversas condições de carga. O administrador poderia executar outras experiências que variam os tempos de serviço para cada serviço, a fim de descobrir a sensibilidade do desempenho a cada um deles. Os parâmetros do sistema também podem ser variados, de modo que o ajuste do desempenho pode ser feito no modelo de simulação.

Benchmarks de desempenho

À medida que os servidores de banco de dados se tornam mais padronizados, o fator diferenciador entre os produtos de diferentes fornecedores é o desempenho destes. Os **benchmarks de desempenho** são pacotes de tarefas que são usadas para quantificar o desempenho dos sistemas de software.

Pacotes de tarefas

Já que a maioria dos sistemas de software, como os bancos de dados, é complexa, há muita variação em sua implementação por diferentes fornecedores. Como resultado, existe uma quantidade de variação significativa em seu desempenho nas diferentes tarefas. Um sistema pode ser o mais eficiente em determinada tarefa; outro pode ser o mais eficiente em uma tarefa diferente. Logo, uma única tarefa normalmente não é suficiente para quantificar o desempenho do sistema. Em vez disso, o desempenho de um sistema é medido pelos pacotes de tarefas padronizadas, chamados *benchmarks de desempenho*.

A combinação dos números de desempenho de várias tarefas precisa ser feita com cuidado. Suponha que tenhamos duas tarefas, T_1 e T_2, e que meçamos o throughput de um sistema como o número de transações de cada tipo que é executada em determinada quantidade de tempo – digamos, 1 segundo. Suponha que o sistema A execute T_1 a 99 transações por segundo e T_2 a 1 transação por segundo. De modo semelhante, considere que o sistema B execute T_1 e T_2 a 50 transações por segundo. Suponha também que uma carga de trabalho tenha uma mistura igual dos dois tipos de transações.

Se apanhássemos a média dos dois pares de números (ou seja, 99 e 1, *versus* 50 e 50), pode parecer que os dois sistemas têm o mesmo desempenho. Porém, é *errado* apanhar as médias dessa forma – se executarmos 50 transações de cada tipo, o sistema *A* levaria cerca de 50,5 segundos para terminar, enquanto o sistema *B* terminaria em apenas 2 segundos!

O exemplo mostra que uma medida simples do desempenho é enganosa se houver mais de um tipo de transação. O modo certo de calcular a média pelos números é apanhar o **tempo para conclusão** da carga de trabalho, em vez do **throughput** médio para cada tipo de transação. Podemos, então, calcular o desempenho do sistema com precisão em transações por segundo para uma carga de trabalho especificada. Assim, o sistema A leva 50,5/100, que é 0,505 segundos por transação, enquanto o sistema B leva 0,02 segundos por transação, na média. Em termos de throughput, o sistema A é executado a uma média de 1,98 transação por segundo, enquanto o sistema B é executado a 50 transações por segundo. Supondo que as transações de todos os tipos sejam igualmente prováveis, a forma correta de calcular a média dos throughputs sobre diferentes tipos de transação é apanhar a **média harmônica** dos throughputs. A média harmônica de n throughputs $t_1, \ldots, t_n$ é definida como

$$\frac{n}{\frac{1}{t_1} + \frac{1}{t_2} + \cdots + \frac{1}{t_n}}$$

Para o nosso exemplo, a média harmônica dos throughputs no sistema A é 1,98. Para o sistema B, ela é 50. Assim, o sistema B é aproximadamente 25 vezes mais rápido do que o sistema A em uma carga de trabalho em uma mistura igual dos dois tipos de exemplo das transações.

Classes de aplicação de banco de dados

Online Transaction Processing (OLTP) e **apoio à decisão** [incluindo o Online Analytical Processing (OLAP)] são duas classes gerais de aplicações tratadas por sistemas de banco de dados. Essas duas classes de tarefas possuem diferentes requisitos. Alta concorrência e técnicas inteligentes para agilizar o processamento de commit são exigidas para dar suporte a uma alta taxa de transações de atualização. Por outro lado, bons algoritmos de avaliação de consulta e otimização de consulta são necessários para o apoio à decisão. A arquitetura de alguns sistemas de banco de dados foi ajustada para o processamento de transação; a de outros, como a série Teradata DBC de sistemas de bancos de dados paralelos, foi ajustada para o apoio à decisão. Outros fornecedores procuram um equilíbrio entre as duas tarefas.

As aplicações normalmente têm uma mistura de requisitos de processamento de transação e apoio à decisão. Logo, qual sistema de banco de dados é melhor para uma aplicação depende de qual mistura dos dois requisitos a aplicação tenha.

Suponha que tenhamos números de throughput para as duas classes de aplicações separadamente, e a aplicação em mãos tenha uma mistura de transações nas duas classes. Temos de ter cuidado até mesmo sobre como tomar a média harmônica dos números de throughput, devido à **interferência** entre as transações. Por exemplo, uma transação de apoio à decisão de longa duração pode adquirir diversos bloqueios, o que pode impedir todo o progresso das transações de atualização. A média harmônica de throughputs deve ser usada apenas se as transações não interferirem umas com as outras.

Os benchmarks TPC

O Transaction Processing Performance Council (TPC) definiu detalhadamente uma série de padrões de benchmark para sistemas de banco de dados.

Os benchmarks definem o conjunto de relações e os tamanhos das tuplas. Eles definem o número de tuplas nas relações não como um número fixo, mas como um múltiplo do número de transações reivindicadas por segundo, para refletir que uma taxa maior de execução de transação provavelmente estará co-relacionada a um número maior de contas. A métrica de desempenho é o throughput, expresso como **transações por segundo** (TPS). Quando seu desempenho é medido, o sistema precisa oferecer um tempo de resposta dentro de certos limites, de modo que um alto throughput não pode ser obtido ao custo de tempos de resposta muito longos. Além do mais, para aplicações de negócios, o custo tem grande importância. Logo, o benchmark TCP também mede o desempenho em termos de **preço por TPS**. Um sistema grande pode ter um alto número de transações por segundo, mas pode ser dispendioso (ou seja, ter um alto preço por TPS). Além do mais, uma empresa não pode reivindicar números de benchmark TCP para os seus sistemas *sem* uma auditoria externa que garanta que o sistema segue fielmente a definição do benchmark, incluindo suporte total para as propriedades ACID das transações.

O primeiro da série foi o benchmark **TPC-A**, definido em 1989. Esse benchmark simula uma aplicação bancária típica por um único tipo de transação que modela o saque e o depósito de dinheiro em um caixa de banco. A transação atualiza várias relações – como o saldo bancário, o saldo do caixa e o saldo do cliente – e acrescenta um registro a uma relação de trilha de auditoria. O benchmark também incorpora a comunicação com terminais, para modelar o desempenho de ponta a ponta do sistema de forma realista. O benchmark **TPC-B** foi projetado para testar o desempenho básico do sistema de banco de dados, junto com o sistema operacional em que o sistema roda. Ele remove as partes do benchmark TPC-A que lidam com os usuários, a comunicação e os terminais, para enfocar o servidor de banco de dados de back-end. Nem TPC-A nem TPC-B são muito usados hoje.

O **benchmark TPC-C** foi projetado para modelar um sistema mais complexo do que o benchmark TPC-A. O benchmark TPC-C se concentra nas principais atividades em um ambiente de entrada de pedido, como entrada e remessa de pedidos, registro de pagamentos, verificação de status de pedidos e monitoração de níveis de estoque. O benchmark TPC-C ainda é muito utilizado para o processamento de transações.

O **benchmark TPC-D** foi projetado para testar o desempenho de sistemas de banco de dados sobre consultas de apoio à decisão. Os sistemas de apoio à decisão estão se tornando cada vez mais importantes. Os benchmarks TPC-A, TPC-B e TPC-C medem o desempenho sobre as cargas de trabalho de processamento de transação, e não devem ser usados como uma medida de desempenho nas consultas de apoio à decisão. O D de TPC-D significa **apoio à decisão**. O esquema de benchmark TPC-D modela uma aplicação de vendas/distribuição, com peças, fornecedores, clientes e pedidos, junto com algumas informações auxiliares. Os tamanhos das relações são definidos como uma razão, e o tamanho do banco de dados é o tamanho total de todas as relações, expresso em gigabytes. TPC-D no fator de escala 1 representa o benchmark TPC-D em um banco de dados de 1 gigabyte, enquanto o fator de escala 10 representa um banco de dados de 10 gigabytes. A carga de trabalho do benchmark consiste em um conjunto de 17 consultas SQL modelando as tarefas comuns executadas nos sistemas de apoio à decisão. Algumas das consultas utilizam recursos SQL complexos, como agregação e consultas aninhadas.

Os usuários do benchmark logo observaram que as diversas consultas TPC-D poderiam ser bastante agilizadas com o uso de visões materializadas e outras informações redundantes. Existem aplicações, como tarefas de relatório periódico, em que as consultas são conhecidas em avanço e a visão materializada pode ser cuidadosamente selecionada para agilizar as consultas. Porém, é necessário levar em conta a sobrecarga de manutenção de visões materializadas.

O **benchmark TPC-R** (em que R significa **relatório**) é uma melhoria do benchmark TPC-D. O esquema é o mesmo, mas existem 22 consultas, das quais 16 são do TPC-D. Além disso, existem duas atualizações, um conjunto de inserções e um conjunto de exclusões. O banco de dados que está realizando o benchmark tem permissão para usar visões materializadas e outras informações redundantes.

Ao contrário, o **benchmark TPC-H** (em que H representa **ad hoc**) utiliza o mesmo esquema e carga de trabalho do TPC-R, mas proíbe visões materializadas e outras informações redundantes, e permite índices apenas sobre as chaves primária e estrangeira. Esse benchmark modela a consulta ocasional, em que as consultas não são conhecidas de antemão, de modo que não é possível criar visões materializadas apropriadas antes da hora.

Tanto TPC-H quanto TPC-R medem o desempenho desta maneira: o **teste de potência** executa as consultas e atualizações uma de cada vez, sequencialmente, e 3.600 segundos divididos pela média geométrica dos tempos de execução das consultas (em segundos) dão uma medida das consultas por hora. O **teste de throughput** executa vários fluxos em paralelo, com cada fluxo executando todas as 22 consultas. Há também um fluxo de atualização paralela. Aqui, o tempo total para a execução inteira é usado para calcular o número de consultas por hora.

A **consulta composta por métrica de hora**, que é a métrica geral, é obtida como a raiz quadrada do produto das métricas de potência e throughput. Uma **métrica composta de preço/desempenho** é definida dividindo-se o preço do sistema pela métrica composta.

O **benchmark TPC-W**, de comércio pela Web, é um benchmark de ponta a ponta que modela sites que contêm conteúdo estático (principalmente imagens) e conteúdo dinâmico gerado a partir de um banco de dados. O caching de conteúdo dinâmico é permitido especificamente, pois é muito útil para agilizar sites. O benchmark modela uma livraria eletrônica, e como outros benchmarks TPC, providencia fatores de escala diferentes. As principais métricas de desempenho são **interações Web por segundo (WIPS – Web Interactions Per Second)** e preço por WIPS.

Os benchmarks OODB

A natureza das aplicações em um banco de dados orientado a objeto (OODB) é diferente daquela das aplicações típicas de processamento de transação. Portanto, um conjunto diferente de benchmarks foi proposto para os OODBs.

O benchmark Object Operations, versão 1, popularmente conhecido como **OO1 benchmark**, foi uma proposta antiga. O **OO7 benchmark** segue uma filosofia diferente daquela dos benchmarks TPC. Os benchmarks TPC oferecem um ou dois números (em termos de transações médias por segundo e transações por segundo por dólar); o benchmark 007 oferece um conjunto de números, contendo um número de benchmark separado para cada um dos vários tipos diferentes de operações. O motivo para essa técnica é que não é claro o que significa a transação OODB *típica*. É claro que tal transação executará certas operações, como a travessia de um conjunto de objetos conectados ou a recuperação de todos os objetos em uma classe, mas não é claro exatamente que mistura dessas operações será utilizada. Logo, o benchmark oferece números separados para cada classe de operações; os números podem ser combinados de uma maneira apropriada, dependendo da aplicação específica.

Padronização

Padrões definem a interface de um sistema de software; por exemplo, os padrões definem a sintaxe e a semântica de uma linguagem de programação, ou as funções em uma interface de programa de aplicação, ou ainda um modelo de dados (como os padrões de banco de dados orientados a objeto). Hoje, os sistemas de banco de dados são complexos e normalmente compostos de várias partes criadas independentemente, que precisam interagir. Por exemplo, os programas do cliente podem ser criados independentemente dos sistemas de back-end, mas os dois precisam ser capazes de interagir entre si. Uma empresa que possui vários sistemas de banco de dados heterogêneos pode precisar trocar dados entre os bancos de dados. Dado tal cenário, os padrões desempenham um papel importante.

Padrões formais são aqueles desenvolvidos por uma organização de padrões ou por grupos do setor, via um processo público. Produtos dominantes às vezes se tornam **padrões de fato**, pois se tornam geralmente aceitos como padrões sem qualquer processo formal de reconhecimento. Alguns padrões formais, como muitos aspectos dos padrões SQL-92 e SQL:1999, são **padrões antecipatórios** que lideram o mercado; eles definem recursos que os fornecedores implementarão nos produtos. Em outros casos, os padrões, ou partes deles, são **padrões reacionários**, pois tentam padronizar recursos que alguns fornecedores já implementaram e que podem ainda ter se tornado padrões de fato. SQL-89 foi de várias maneiras um padrão reacionário, pois padronizou recursos, como verificação de integridade, que já estavam presentes no padrão IBM SAA SQL e em outros bancos de dados.

Comitês de padrões formais normalmente são compostos de representantes dos fornecedores e dos membros dos grupos de usuários e organizações de padrões, como o International Organization for Standardization (ISO), o American National Standards Institute (ANSI) ou agências profissionais, como o Institute of Electrical and Electronics Engineers (IEEE). Comitês de padrões formais se reúnem periodicamente, e os membros apresentam propostas para recursos serem adicionados ou modificados no padrão. Depois de um período de discussão (normalmente estendido), modificações à proposta e crítica pública, os membros votam se aceitarão ou rejeitarão um recurso. Algum tempo depois de um padrão ter sido definido e implementado,

suas limitações se tornam claras, e novos requisitos se tornam aparentes. O processo de atualização do padrão começa, e uma nova versão do padrão normalmente é liberada depois de alguns anos. Esse ciclo normalmente se repete depois de alguns anos, até que por fim (talvez muitos anos depois) o padrão se torna tecnologicamente irrelevante, ou perde sua base de usuários.

O padrão DBTG CODASYL para bancos de dados em rede, formulado pela Database Task Group, foi um dos primeiros padrões formais para bancos de dados. Os produtos de banco de dados da IBM estabeleceram inicialmente os padrões de fato, pois a IBM comandava grande parte do mercado de banco de dados. Com o crescimento dos bancos de dados relacionais chegou uma série de novos participantes no negócio de banco de dados; daí a necessidade de padrões formais. Nos últimos anos, a Microsoft criou uma série de especificações que também se tornaram padrões de fato. Um exemplo notável é o ODBC, que agora é usado nos ambientes não Microsoft. JDBC, cuja especificação foi criada pela Sun Microsystems, é outro padrão de fato bastante utilizado.

Esta seção oferece uma visão bem geral dos diferentes padrões, concentrando-se nos objetivos do padrão. As notas bibliográficas ao final do capítulo oferecem referências a descrições detalhadas dos padrões mencionados nesta seção.

Padrões SQL

Como SQL é a linguagem de consulta mais utilizada, muito trabalho foi feito na sua padronização. ANSI e ISO, com os diversos fornecedores de banco de dados, desempenharam um papel importante nesse trabalho. O padrão SQL-86 foi a versão inicial. O padrão IBM Systems Application Architecture (SAA) para SQL foi lançado em 1987. À medida que as pessoas identificaram a necessidade de mais recursos, versões atualizadas do padrão SQL formal foram desenvolvidas, chamadas SQL-89 e SQL-92.

A versão SQL:1999 do padrão SQL acrescentou uma série de recursos à linguagem. Vimos muitos desses recursos nos primeiros capítulos. A versão SQL:2003 do padrão SQL é uma extensão secundária do padrão SQL:1999. Alguns recursos, como os recursos de OLAP da SQL:1999 (seção "Agregação estendida" do Capítulo 18) foram especificados como um adendo à versão anterior do padrão SQL:1999, em vez de esperar pelo lançamento da SQL:2003.

O padrão SQL:2003 é dividido em várias partes:

- Parte 1: SQL/Framework oferece uma visão geral do padrão.
- Parte 2: SQL/Foundation define os fundamentos do padrão: tipos, esquemas, tabelas, visões, instruções de consulta e atualização, expressões, modelo secundário, predicados, regras de atribuição, gerenciamento de transação, e assim por diante.
- Parte 3: SQL/CLI (Call Level Interface) define interfaces de programa de aplicação para a SQL.
- Parte 4: SQL/PSM (Persistent Stored Modules) define extensões à SQL para torná-la procedural.
- Parte 9: SQL/MED (Management of External Data) define padrões ou interfaces de um sistema SQL às origens externas. Escrevendo wrappers, os projetistas do sistema podem tratar de origens de dados externas, como arquivos ou dados em bancos de dados não relacionais, como se fossem tabelas "estrangeiras".
- Parte 10: SQL/OLB (Object Language Bindings) define padrões para embutir a SQL na Java.
- Parte 11: SQL/Schemata (Information and Definition Schema) define uma interface padrão de catálogo.
- Parte 13: SQL/JRT (Java Routines and Types) define padrões para acessar rotinas e tipos em Java.
- Parte 14: SQL/XML define especificações relacionadas a XML.

Os números que faltam abrangem recursos como dados temporais, processamento de transações distribuído e dados de multimídia, para os quais ainda não existe um consenso sobre os padrões.

Padrões de conectividade de banco de dados

O padrão ODBC é muito usado para a comunicação entre aplicações cliente e sistemas de banco de dados. ODBC é baseado nos padrões SQL **Call Level Interface (CLI)** desenvolvidos pelo consórcio do setor *X/Open* e pelo SQL Access Group, mas possui várias extensões. A API ODBC define uma CLI, uma definição de sintaxe SQL e regras sobre seqüências permissíveis de chamadas CLI. O padrão também define níveis de conformidade para a CLI e a sintaxe da SQL. Por exemplo, o nível central da CLI possui comandos para a conexão com um banco de dados, para preparar e executar instruções SQL, obter resultados e valores de status de volta, e ainda gerenciar transações. O próximo nível de conformidade (nível 1) requer suporte para recuperação de informações de catálogo e alguns outros recursos acima do nível CLI básico; o nível 2 requer outros recursos, como a capacidade de enviar e apanhar arrays de valores de parâmetro e obter informações de catálogo mais detalhadas.

ODBC permite que um cliente se conecte simultaneamente a várias origens de dados a fim de alterar entre elas, mas as transações em cada uma são independentes; ODBC não admite o commit em duas fases.

Um sistema distribuído oferece um ambiente mais geral do que um sistema cliente-servidor. O consórcio X/Open também desenvolveu os **padrões X/Open XA** para interoperação de bancos de dados. Esses padrões definem primitivas de gerenciamento de transação (como transaction begin, commit, abort e prepare-to-commit) que os bancos de dados compatíveis deverão oferecer; um gerenciador de transação pode invocar essas primitivas para implementar transações distribuídas por commit de duas fases. Os padrões XA são independentes do modelo de dados e das interfaces específicas entre clientes e bancos de dados para troca de dados. Assim, podemos usar os protocolos XA para implementar um sistema de transação distibuído, em que uma transação do sistema pode acessar bancos de dados relacionais e também orientados a objeto, mas o gerenciador de transações garante a consistência global via commit de duas fases.

Existem muitas origens de dados que não são bancos de dados relacionais, e de fato podem nem sequer ser bancos de dados. Alguns exemplos são arquivos planos e depósitos de e-mail. O **OLE-DB** da Microsoft é uma API C++ com objetivos semelhantes ao ODBC, mas para origens de dados não de banco de dados, que pode oferecer facilidades limitadas de consulta e atualização. Assim como ODBC, OLE-DB oferece construções para conectar a uma origem de dados, iniciar uma sessão, executar comandos e receber resultados de volta na forma de um rowset, que é um conjunto de linhas de resultado.

Porém, OLE-DB difere de ODBC de várias maneiras. Para dar suporte a origens de dados com suporte limitado de recursos, os recursos em OLE-DB são divididos em uma série de interfaces, e uma origem de dados pode implementar somente um subconjunto das interfaces. Um programa OLE-DB pode negociar com uma origem de dados para descobrir quais interfaces são aceitas. Em ODBC, os comandos sempre estão em SQL. Em OLE-DB, os comandos podem estar em qualquer linguagem aceita pela origem de dados; enquanto algumas origens podem ter suporte para SQL, ou a um subconjunto limitado da SQL, outras origens podem oferecer apenas capacidades simples, como acesso a dados em um arquivo plano, sem qualquer capacidade de consulta. Outra diferença importante entre OLE-DB e ODBC é que um rowset é um objeto que pode ser compartilhado por várias aplicações por meio da memória compartilhada. Um objeto rowset pode ser atualizado por uma aplicação, e outras aplicações compartilhando esse objeto seriam notificadas sobre a mudança.

A API **Active Data Objects (ADO)**, também criada pela Microsoft, oferece uma interface de fácil utilização para a funcionalidade OLE-DB, que pode ser chamada a partir de linguagens de scripting, como VBScript e JScript. A API ADO.NET mais recente foi projetada para aplicações escritas nas linguagens .NET, como C# e Visual Basic.NET. Além de oferecer interfaces simplificadas, ela oferece uma abstração chamada *DataSet*, que permite o acesso desconectado aos dados.

Padrões de banco de dados de objeto

Os padrões na área de bancos de dados orientados a objeto até aqui foram escritos principalmente por fornecedores de OODB. O *Object Database Management Group* (ODMG) foi um grupo formado por fornecedores de OODB para padronizar o modelo de dados e as interfaces de linguagem aos OODBs. A interface da linguagem C++ especificada pelo ODMG foi rapidamente explicada no Capítulo 9. ODMG não está mais ativo. JDO é um padrão para acrescentar persistência à Java.

O *Object Management Group* (OMG) é um consórcio de empresas, formado com o objetivo de desenvolver uma arquitetura padrão para aplicações de software distribuídas, baseadas no modelo orientado a objeto. O OMG tornou público o modelo de referência *Object Management Architecture* (OMA). O *Object Request Broker* (ORB) é um componente da arquitetura OMA que oferece despacho de mensagens para objetos distribuídos transparentemente, de modo que o local físico do objeto não é importante. A Common Object Request Broker Architecture (CORBA) oferece uma especificação detalhada do ORB e inclui uma **Interface Description Language (IDL)**, que é usada para definir os tipos de dados usados para intercâmbio de dados. A IDL ajuda a dar suporte à conversão de dados quando estes são enviados entre sistemas com diferentes representações de dados.

Padrões baseados em XML

Uma grande variedade de padrões baseados em XML (ver Capítulo 10) tem sido definida para uma grande variedade de aplicações. Muitos desses padrões estão relacionados a e-commerce. Eles incluem padrões promulgados por consórcios não lucrativos e esforços de apoio corporativo para criar padrões de fato.

RosettaNet, que se encontra na categoria anterior, é um consórcio do setor que usa padrões baseados em XML para facilitar o gerenciamento da cadeia de suprimentos nos setores de computador e tecnologia da informação. O gerenciamento da cadeia de suprimentos refere-se às compras de materiais e serviços de que uma organização precisa para funcionar. Ao contrário, o gerenciamento de relacionamento com o cliente refere-se ao front-end da interação de uma empresa, lidando com os clientes. O gerenciamento da cadeia de suprimentos requer padronização de uma série de itens, como:

- **Identificador global de empresa:** RosettaNet especifica um sistema para identificar empresas de forma exclusiva, usando um identificador de 9 dígitos chamado *Data Universal Numbering System* (DUNS).
- **Identificador global de produto:** RosettaNet especifica um *Global Trade Item Number* (GTIN) de 14 dígitos para identificar produtos e serviços.
- **Identificador global de classe:** Esse é um código hierárquico de 10 dígitos para classificar produtos e serviços, chamado *United Nations/Standard Product and Services Code* (UN/SPSC).
- **Interfaces entre parceiros comerciais:** RosettaNet *Partner Interface Processes* (PIPs) definem processos de negócios entre parceiros. PIPs são diálogos baseados em XML de sistema para sistema: eles definem os formatos e a semântica dos documentos de negócios envolvidos no processo e as etapas envolvidas na realização de uma transação. Alguns exemplos de etapas poderiam incluir obter informações de produto e serviço, ordens de compra, faturamento de pedido, pagamento, solicitações de status do pedido, controle de estoque, suporte pós-vendas, incluindo garantia de serviço, e assim por diante. A troca de informações de projeto, configuração, processo e qualidade também é possível para coordenar atividades de manufatura entre organizações.

Os participantes dos mercados eletrônicos podem armazenar dados em diversos sistemas de banco de dados. Esses sistemas utilizam diferentes modelos, formatose tipos de dados. Além do mais, pode haver diferentes semânticas (medida métrica *versus inglesa*, moedas distintas, e assim por diante) nos dados. Os padrões para mercados eletrônicos incluem métodos para *embrulhar* cada um desses sistemas heterogêneos com um esquema XML. Esses *wrappers* XML formam a base de uma visão unificada dos dados por todos os participantes do mercado.

Simple Object Access Protocol (SOAP) é um padrão de chamada de procedimento remoto que usa XML para codificar dados (tanto parâmetros quanto resultados) e utiliza HTTP como protocolo de transporte; ou seja, uma chamada de procedimento se torna uma solicitação HTTP. SOAP tem o apoio do World Wide Web Consortium (W3C) e obteve grande aceitação no setor. SOAP pode ser usado em diversas aplicações. Por exemplo, no e-commerce de empresa para empresa, as aplicações rodando em um site podem acessar dados e executar ações em outros sites por meio do SOAP.

SOAP e Web services foram descritos com mais detalhes na seção "Web services" do Capítulo 10.

Migração de aplicações

Sistemas legados são sistemas de aplicação de geração mais antiga, que estão em uso por uma organização, mas que a organização deseja substituir por uma aplicação diferente. Por exemplo, muitas organizações desenvolveram aplicações em casa, mas podem decidir substituí-las por um produto comercial. Em alguns casos, um sistema legado pode usar a tecnologia antiga, que é incompatível com padrões e sistemas da geração atual. Alguns sistemas legados em operação hoje já têm várias décadas de uso e são baseados em tecnologias como bancos de dados que usam modelos de dados de rede ou hierárquicos, ou utilizam Cobol e sistemas de arquivos sem um banco de dados. Esses sistemas ainda podem conter dados valiosos e dar suporte a aplicações críticas.

A substituição de uma aplicação legada por uma nova aplicação normalmente é dispendiosa em termos de tempo e de dinheiro, pois geralmente são muito grandes, consistindo em milhões de linhas de código desenvolvidas por equipes de programadores, geralmente durante várias décadas. Elas contêm grandes quantidades de dados e precisam ser portadas para a nova aplicação, que podem usar um esquema completamente diferente. A passagem de uma aplicação antiga para uma nova envolve o novo treinamento de muitas pessoas. A passagem normalmente precisa ser feita sem qualquer interrupção, com os dados inseridos no sistema antigo disponíveis também por meio do novo sistema.

Muitas organizações tentam evitar a substituição de sistemas legados e, em vez disso, tentam interoperá-las com os sistemas mais novos. Uma técnica utilizada para interoperar entre bancos de dados relacionais e bancos de dados legados é criar uma camada, chamada **wrapper**, em cima dos sistemas legados que podem fazer com que o sistema legado pareça ser um banco de dados relacional. O wrapper pode oferecer suporte para ODBC ou outros padrões de interconexão como OLE-DB, que pode ser usado para consultar e atualizar o sistema legado. O wrapper é responsável por converter consultas relacionais e atualizações para consultas e atualizações no sistema legado.

Quando uma organização decide substituir um sistema legado por um novo, pode seguir um processo chamado **engenharia reversa**, que consiste em passar pelo código do sistema legado para chegar a projetos de esquema no modelo de dados exigido (como um modelo ER ou um modelo de dados orientado a objeto). A engenharia reversa também examina o código para descobrir que procedimentos e processos foram implementados, a fim de obter um modelo de alto nível do sistema. A engenharia reversa é necessária porque os sistemas legados normalmente não possuem documentação de alto nível de seu esquema e projeto geral do sistema. Quando surgem com o projeto de um novo sistema, os desenvolvedores revisam o projeto, de modo que possa ser melhorado, em vez de apenas reimplementado como se encontra. A codificação extensa

é necessária para dar suporte a toda a funcionalidade (como a interface com o usuário e os sistemas de relatório) que foi fornecida pelo sistema legado. O processo geral é denominado **reengenharia**.

Quando um novo sistema tiver sido montado e testado, o sistema precisa ser preenchido com dados do sistema legado, e todas as atividades precisam ser executadas no novo sistema. Contudo, a transição brusca para um novo sistema, que é chamada de **técnica big-bang**, carrega diversos riscos. Primeiro, os usuários podem não estar familiarizados com as interfaces do novo sistema. Segundo, pode haver bugs ou problemas de desempenho no novo sistema, que não foram descobertos quando ele foi testado. Esses problemas podem levar a grandes perdas para as empresas, pois sua capacidade de executar transações críticas, como vendas e compras, pode ser severamente afetada. Em alguns casos extremos, o novo sistema foi até mesmo abandonado, e o sistema legado reutilizado, após uma tentativa de passagem fracassada.

Uma técnica alternativa, chamada **técnica chicken-little**, substitui de forma incremental a funcionalidade do sistema legado. Por exemplo, as novas interfaces do usuário podem ser usadas com o sistema antigo no back-end, ou vice-versa. Outra opção é usar o novo sistema somente para alguma funcionalidade que possa ser desacoplada do sistema legado. De qualquer forma, os sistemas legado e novo coexistem por algum tempo. Portanto, há uma necessidade de desenvolver e usar wrappers no sistema legado a fim de oferecer a funcionalidade exigida para interoperar com o novo sistema. Essa técnica, portanto, tem um custo de desenvolvimento maior associado a ela.

Resumo

- Ajustar os parâmetros do sistema de banco de dados, bem como o projeto de banco de dados de nível mais alto – como o esquema, índices e transações – é importante para o bom desempenho. O ajuste é melhor quando se identificam e eliminam os gargalos.
- Os benchmarks de desempenho possuem um papel importante nas comparações de sistemas de banco de dados, especialmente quando os sistemas se tornam mais compatíveis com os padrões. Os pacotes de benchmark TPC são muito usados, e os diferentes benchmarks TPC são úteis para comparação do desempenho dos bancos de dados sob diferentes cargas de trabalho.
- Os padrões são importantes por causa da complexidade dos sistemas de banco de dados e de sua necessidade de interoperação. Existem padrões formais para SQL. Os padrões de fato, como ODBC e JDBC, e os padrões adotados pelos grupos do setor, como CORBA, desempenharam um papel importante no crescimento dos sistemas de banco de dados cliente-servidor. Padrões para bancos de dados orientados a objeto, como ODMG, estão sendo desenvolvidos por grupos do setor.
- Sistemas legados são sistemas baseados em tecnologias de geração mais antiga, como bancos de dados não relacionais ou mesmo diretamente nos sistemas de arquivo. A interface de sistemas legados com sistemas da nova geração normalmente é importante quando executam sistemas de missão crítica. A migração de sistemas legados para sistemas da nova geração precisa ser feita cuidadosamente, de modo a evitar interrupções, que podem ser muito dispendiosas.

Termos de revisão

- Ajuste de desempenho
- Gargalos
- Sistemas de enfileiramento
- Parâmetros ajustáveis
- Ajuste de hardware
- Regra dos cinco minutos
- Regra do um minuto
- Ajuste do esquema
- Ajuste de índices
- Visões materializadas
- Manutenção de visão imediata
- Manutenção de visão adiada
- Ajuste de transações
- Melhorando a orientação a conjunto
- Transações de minibatch
- Simulação de desempenho
- Benchmarks de desempenho
- Tempo de serviço
- Tempo para conclusão
- Classes de aplicação de banco de dados
- Os benchmarks TPC
  - TPC-A
  - TPC-B
  - TPC-C
  - TPC-D
  - TCP-R
  - TCP-H
  - TPC-W
- Interações Web por segundo
- Benchmarks OODB
  - OO1
  - OO7
- Padronização
  - Padrões formais
  - Padrões de fato
  - Padrões antecipatórios
  - Padrões reacionários

- Padrões de conectividade de banco de dados
  - ODBC
  - OLE-DB
  - Padrões X/Open XA
- Padrões de banco de dados de objeto
  - ODMG
  - CORBA
- Padrões baseados em XML
- Sistemas legados
- Engenharia reversa
- Reengenharia

Exercícios práticos

23.1 Muitas aplicações precisam gerar números de seqüência para cada transação.

a. Se um contador de seqüência estiver bloqueado de uma maneira de duas fases, ele pode se tornar um gargalo de concorrência. Explique por que isso pode acontecer.

b. Muitos sistemas de banco de dados admitem contadores de seqüência embutidos, que não estão bloqueados de uma maneira de duas fases; quando uma transação solicita um número de seqüência, o contador está bloqueado, incrementado e desbloqueado.

i. Explique como esses contadores podem melhorar a concorrência.

ii. Explique por que pode haver lacunas nos números de seqüência pertencentes ao conjunto final de transações confirmadas.

23.2 Suponha que você receba uma relação $r(a, b, c)$.

a. Dê um exemplo de uma situação em que o desempenho das consultas de seleção de igualdade sobre o atributo a possa ser bastante afetado pelo modo como r é agrupado.

b. Suponha que você também tivesse consultas de seleção de intervalo sobre o atributo b. Você poderia agrupar r de modo que as consultas de seleção de igualdade sobre $r.a$ e as consultas de seleção de intervalo sobre $r.b$ pudessem ser respondidas de modo eficiente? Explique sua resposta.

c. Se esse agrupamento não for possível, sugira como os dois tipos de consultas podem ser executados de forma eficiente escolhendo índices apropriados, supondo que o seu banco de dados admita planos apenas de índice (ou seja, se todas as informações exigidas para uma consulta estiverem disponíveis em um índice, o banco de dados pode gerar um plano que usa o índice mas não acessa a relação).

23.3 Suponha que uma aplicação de banco de dados parece não ter um único gargalo, ou seja, CPU e utilização de disco são altos, e todas as filas de banco de dados são aproximadamente balanceadas. Isso significa que a aplicação não pode ser mais ajustada? Explique a sua resposta.

23.4 Suponha que um sistema execute três tipos de transações. Transações do tipo A são executadas a uma taxa de 50 por segundo, transações do tipo B são executadas a 100 por segundo, e transações do tipo C são executadas a 200 por segundo. Suponha que a mistura de transações tenha 25% do tipo A, 25% do tipo B e 50% do tipo C.

a. Qual é o throughput de transação médio do sistema, supondo que não exista interferência entre as transações?

b. Que fatores podem resultar na interferência entre as transações de diferentes tipos, fazendo com que o throughput calculado seja incorreto?

23.5 Liste alguns benefícios e desvantagens de um padrão antecipatório em comparação com um padrão reacionário.

Exercícios

23.6 Descubra toda a informação de desempenho que seu sistema de banco de dados favorito oferece. Procure pelo menos o seguinte: que consultas estão atualmente sendo executadas ou foram executadas recentemente, que recursos cada uma delas consumiu (CPU e E/S), que fração de solicitações de página resultaram em perdas de buffer (para cada consulta, se estiver disponível) e que bloqueios possuem um alto grau de disputa. Você também pode ser capaz de obter informações sobre utilização de CPU e E/S do sistema operacional.

23.7 a. Quais são os três níveis gerais em que um sistema de banco de dados pode ser ajustado para melhorar o desempenho?

b. Dê dois exemplos de como o ajuste pode ser feito, para cada um dos níveis.

23.8 Ao executar o ajuste do desempenho, você deverá tentar ajustar seu hardware (acrescentando discos ou memória) primeiro, ou tentar ajustar suas transações (acrescentando índices ou visões materializadas) primeiro? Explique a sua resposta.

23.9 Suponha que a sua aplicação tenha transações que acessem e atualizem uma única tupla em uma relação muito grande, armazenada em uma organização de arquivo de árvore B+. Suponha que todos os nós internos da árvore B+ estejam na memória, mas somente uma fração muito pequena das páginas de folha

possa caber na memória. Explique como calcular o número mínimo de discos exigidos para admitir uma carga de trabalho de 100 transações por segundo. Calcule também o número necessário de discos, usando valores para os parâmetros de disco dados na seção "Discos magnéticos" do Capítulo 11.

23.10 Qual é a motivação para dividir uma transação longa em uma série de transações pequenas? Que problemas poderiam surgir como resultado, e como esses problemas podem ser evitados?

23.11 Suponha que o preço da memória caia para a metade, e a velocidade do acesso ao disco (número de acessos por segundo) dobre, enquanto todos os outros fatores permanecem inalterados. Qual seria o efeito dessa mudança nas regras de 5 minutos e 1 minuto?

23.12 Liste pelo menos 4 recursos dos benchmarks TPC que ajudam a torná-los medidas realistas e dignas de confiança.

23.13 Por que o benchmark TPC-D foi substituído pelos benchmarks TPC-H e TPC-R?

23.14 Explique que características da aplicação o ajudariam a decidir qual dentre TPC-C, TCP-H e TPC-R modela melhor a aplicação.

Notas bibliográficas

Uma antiga proposta para um benchmark de sistema de banco de dados (o benchmark Wisconsin) foi feita por Bitton *et al.* [1983]. Os benchmarks TPC-A, B e C são descritos em Gray [1991]. Uma versão on-line de todas as descrições de benchmark TPC, bem como os resultados do benchmark, está disponível na World Wide Web no URL `www.tpc.org`; o site também contém informações atualizadas sobre novas propostas de benchmark. Poess e Floyd [2000] oferecem uma visão geral dos benchmarks TPC-H, TPC-R e TPC-W. O benchmark OO1 para OODBs é descrito em Cattell e Skeen [1992]; o benchmark OO7 é descrito em Carey *et al.* [1993].

Kleinrock [1975] e Kleinrock [1976] é um livro-texto popular em dois volumes, sobre teoria de enfileiramento.

Shasha e Bonnet [2002] oferecem cobertura detalhada do ajuste de banco de dados. O'Neil e O'Neil [2000] oferecem um livro-texto muito bom sobre medição e ajuste de desempenho. As regras de 5 minutos e 1 minuto são descritas em Gray e Putzolu [1987] e Gray e Graefe [1997]. A seleção de índice e a seleção de visão materializada são explicadas por Ross *et al.* [1996], Labio *et al.* [1997], Gupta [1997], Chaud-huri e Narasayya [1997], Agrawal *et al.* [2000] e Mistry *et al.* [2001]. Zilio *et al.* [2004], Dageville *et al.* [2004a] e Agrawal *et al.* [2004] descrevem o suporte para ajuste no IBM DB2, Oracle e Microsoft SQL Server.

Para obter referências sobre os padrões SQL, consulte as notas bibliográficas do Capítulo 3.

Informações sobre ODBC, OLE-DB, ADO e ADO.NET podem ser encontradas no site `www.microsoft.com/data` e em diversos livros sobre o assunto, que podem ser encontrados em `www.amazon.com`. *ACM Sigmod Record*, que é publicado trimestralmente, tem uma seção regular sobre padrões em bancos de dados.

Muitas informações sobre padrões e ferramentas baseadas em XML estão disponíveis on-line no site `www.w3c.org`. Informações sobre RosettaNet podem ser encontradas na Web em `www.rosettanet.org`.

A reengenharia do processo comercial é explicada por Cook [1996]. Umar [1997] explica a reengenharia e as questões referentes a sistemas legados.

CAPÍTULO 24

Tipos de dados avançados e novas aplicações

Durante a maior parte do história dos bancos de dados, os tipos de dados armazenados nos bancos de dados foram relativamente simples, e isso foi refletido no suporte um tanto limitado para os tipos de dados nas versões anteriores da SQL. Nos últimos anos, porém, houve uma necessidade maior para lidar com novos tipos de dados nos bancos de dados, como dados temporais, dados espaciais e dados de multimídia.

Outra tendência importante na última década criou seus próprios problemas: o crescimento de computadores móveis, começando com computadores de laptop e organizadores de bolso e estendendo nos anos mais recentes a telefones móveis com computadores embutidos e uma série de *computadores* de vestir que são cada vez mais usados nas aplicações comerciais.

Neste capítulo, estudamos vários tipos novos e também estudamos as questões de banco de dados que lidam com computadores móveis.

Motivação

Antes de entrarmos em cada um dos assuntos com detalhes, resumimos a motivação e algumas questões importantes ao lidar com cada um desses tipos de dados.

- **Dados temporais.** A maioria dos sistemas de banco de dados modela o estado atual do mundo, por exemplo, clientes atuais, alunos atuais e cursos atualmente sendo oferecidos. Em muitas aplicações, é muito importante armazenar e obter informações sobre os estados anteriores. As informações históricas podem ser incorporadas manualmente em um projeto de esquema. Porém, a tarefa é bastante simplificada pelo suporte do banco de dados para dados temporais, que estudamos na próxima seção.
- **Dados espaciais.** Os dados espaciais incluem **dados geográficos**, como mapas e informações associadas, e **dados de projeto auxiliado por computador**, como projetos de circuito integrado ou projetos de prédios. As aplicações de dados espaciais inicialmente armazenavam dados como arquivos em um sistema de arquivos, como as aplicações comerciais de gerações anteriores. No entanto, a medida que a complexidade, o volume dos dados e o número de usuários cresceram, as técnicas ocasionais para armazenar e recuperar dados em um sistema de arquivos provaram ser insuficientes para as necessidades de muitas aplicações que utilizam dados espaciais.

  As aplicações de dados espaciais exigem facilidades oferecidas por um sistema de banco de dados – em particular, a capacidade de armazenar e consultar grandes quantidades de dados de modo eficiente. Algumas aplicações também podem exigir outros recursos de banco de dados, como atualizações atômicas a partes dos dados armazenados, durabilidade e controle de concorrência. Na seção "Dados espaciais e geográficos", estudamos as extensões necessárias aos sistemas de banco de dados tradicionais para dar suporte a dados espaciais.
- **Dados de multimídia.** Na seção "Bancos de dados de multimídia", estudamos os recursos exigidos nos sistemas de banco de dados que armazenam dados de multimídia como imagem, vídeo e dados de áudio. O principal recurso de distinção de dados de vídeo e áudio é que a tela dos dados exige recuperação em uma velocidade constante e predeterminada; logo, esses dados são denominados **dados de mídia contínua**.
- **Bancos de dados móveis.** Na seção "Mobilidade e bancos de dados pessoais", estudamos os requisitos de banco de dados da nova geração de sistemas de computação

móveis, como computadores notebook e dispositivos de computação palmtop, que estão conectados a estações base por meio de redes de comunicação digital sem fio. Esses computadores precisam ser capazes de operar enquanto estão desconectados da rede, ao contrário dos sistemas de banco de dados distribuídos, discutidos no Capítulo 22. Eles também têm capacidade de armazenamento limitada, e por isso exigem técnicas especiais para gerenciamento de memória.

Tempo nos bancos de dados

Um banco de dados modela o estado de algum aspecto do próprio mundo real exterior. Normalmente, os bancos de dados modelam apenas um estado – o estado atual – do mundo real, e não armazenam informações sobre estados passados, exceto, talvez, como trilhas de auditoria. Quando o estado do mundo real muda, o banco de dados é atualizado, e as informações sobre o estado antigo são perdidas. Porém, em muitas aplicações, é importante armazenar e recuperar informações sobre estados passados. Por exemplo, um banco de dados de pacientes precisa armazenar informações sobre o histórico médico de um paciente. Um sistema de monitoração de fábrica pode armazenar informações sobre as leituras atual e passada dos sensores na fábrica, para análise. Os bancos de dados que armazenam informações sobre estados do mundo real com o passar do tempo são chamados **bancos de dados temporais**.

Ao considerar a questão do tempo nos sistemas de banco de dados, temos de distinguir entre o tempo medido pelo sistema e o tempo observado no mundo real. O **tempo válido** para um fato é o conjunto de intervalos de tempo durante o qual o fato é verdadeiro no mundo real. O **tempo da transação** para um fato é o intervalo de tempo durante o qual o fato é atual dentro do sistema de banco de dados. Esse último tempo é baseado na ordem de seriação de transações e é gerado automaticamente pelo sistema. Observe que os intervalos de tempo válidos, sendo um conceito do mundo real, não podem ser gerados automaticamente e precisam ser fornecidos ao sistema.

Uma **relação temporal** é aquela em que cada tupla possui um tempo associado quando ela é verdadeira; o tempo pode ser o tempo válido ou o tempo da transação. Naturalmente, o tempo válido e o tempo da transação podem ser armazenados, caso em que a relação é considerada uma **relação bitemporal**. A Figura 24.1 mostra um exemplo de uma relação temporal. Para simplificar a representação, cada tupla tem apenas um intervalo de tempo associado a ela; assim, uma tupla é representada uma vez para cada intervalo distinto de tempo em que é verdadeira. Os intervalos são mostrados aqui como um par de atributos *de* e *até*; uma implementação real teria um tipo estruturado, talvez chamado *Intervalo*, que contém os dois campos. Observe que algumas das tuplas possuem um "*" na coluna de tempo *até*; esses asteriscos indicam que a tupla é verdadeira até que o valor na coluna de tempo *até* seja alterado; assim, a tupla é verdadeira na hora atual. Embora os tempos sejam mostrados em formato textual, eles são armazenados internamente em uma forma mais compacta, como o número de segundos desde algum tempo fixo em uma data fixa (como 12:00 A.M., January 1,1900) que pode ser traduzido de volta para a forma textual normal.

Especificação de tempo em SQL

O padrão SQL define os tipos date, time e timestamp. O tipo date contém quatro dígitos para o ano (1-9999), dois dígitos para o mês (1-12) e dois dígitos para a data (1-31). O tipo time contém dois dígitos para a hora, dois dígitos para o minuto e dois dígitos para o segundo, mais dígitos fracionários opcionais. O campo de segundos pode ir além de 60, de modo a permitir os segundos que são acrescentados durante alguns anos para corrigir pequenas variações na velocidade de rotação da Terra. O tipo timestamp contém os campos de data e hora, com seis dígitos fracionários para o campo de segundos.

Como diferentes lugares no mundo possuem diferentes horas locais, normalmente existe a necessidade de especificar o fuso horário junto com a hora. O Universal Coordinated Time (UTC) é um ponto de referência padrão para es-

numero_conta	*nome_agência*	*saldo*	*de*	*até*
A-101	Downtown	500	1999/1/1 9:00	1999/1/24 11:30
A-101	Downtown	100	1999/1/24 11:30	*
A-215	Mianus	700	2000/6/2 15:30	2000/8/8 10:00
A-215	Mianus	900	2000/8/8 10:00	2000/9/5 8:00
A-215	Mianus	700	2000/9/5 8:00	*
A-217	Brighton	750	1999/7/5 11:00	2000/5/1 16:00

Figura 24.1 Uma relação *conta* temporal.

pecificar a hora, com tempos locais definidos como deslocamentos a partir do UTC. (A abreviação padrão é UTC, e não UCT, pois é uma abreviação de "Universal Coordinated Time" escrito em francês como *universel temps coordonné*.) SQL também admite dois tipos, **time com fuso horário** e **timestamp com fuso horário**, que especificam a hora como a hora local mais o deslocamento da hora local desde o UTC. Por exemplo, o horário poderia ser expresso em termos do Eastern Standard Time nos Estados Unidos, com um deslocamento de 6:00, pois o horário Eastern Standard dos Estados Unidos está 6 horas antes da UTC.

A SQL admite um tipo chamado **interval**, que permite nos referirmos a um período de tempo como "1 day" ou "2 days and 5 hours", sem especificar determinada hora quando esse período começa. Essa noção difere da noção de intervalo que usamos anteriormente, que se refere a um intervalo de tempo com horas inicial e final específicas.[1]

Linguagens de consulta temporal

Uma relação de banco de dados sem informações temporais às vezes é chamada de relação de snapshot, pois reflete o estado em um instantâneo do mundo real. Assim, um instantâneo de uma relação temporal em um ponto no tempo t é o conjunto de tuplas na relação que são verdadeiras no momento t, com os atributos de intervalo de tempo removidos. A operação de snapshot sobre uma relação temporal oferece o snapshot da relação em um momento especificado (ou no momento atual, se o tempo não for especificado).

Uma seleção temporal é aquela que envolve os atributos de tempo; uma projeção temporal é uma projeção em que as tuplas na projeção herdam seus tempos das tuplas na relação original. Uma junção temporal é uma junção, com o tempo de uma tupla no resultado sendo a interseção dos tempos das tuplas das quais ela é derivada. Se os itens não tiverem interseção, a tupla é removida do resultado.

Os predicados precedes, overlaps e contains podem ser aplicados aos intervalos; seus significados são: precede, sobrepõe-se parcialmente e contém . A operação de interseção pode ser aplicada a dois intervalos, para dar um único intervalo (possivelmente vazio). Porém, a união de dois intervalos pode ou não ser um único intervalo.

As dependências funcionais precisam ser usadas com cuidado em uma relação temporal. Embora o número de conta possa determinar funcionalmente o saldo em qualquer ponto do tempo, obviamente o saldo pode mudar com o tempo. Uma dependência funcional temporal $X \xrightarrow{T} Y$ se mantém em um esquema de relação R se, para todas as instâncias legais r de R, todos os snapshots de r satisfizerem a dependência funcional $X \rightarrow Y$.

Várias propostas foram feitas para a extensão da SQL a fim de melhorar seu suporte de dados temporais, mas, pelo menos até a versão 2003, a SQL não forneceu qualquer suporte especial para dados temporais além dos tipos de dados e operações relacionadas a tempo.

Dados espaciais e geográficos

O suporte a dados espaciais nos bancos de dados é importante para o armazenamento, indexação e consulta eficientes dos dados com base em locais espaciais. Por exemplo, suponha que queiramos armazenar um conjunto de polígonos em um banco de dados e consultar o banco de dados para encontrar todos os polígonos que fazem a interseção em determinado polígono. Não podemos usar estruturas de índice padrão, como árvores B ou índices de hash, para responder tal consulta de modo eficiente. O processamento eficiente dessa consulta exigiria estruturas de índice de uso especial, como árvores R (que estudamos mais tarde) para a tarefa.

Dois tipos de dados espaciais são particularmente importantes:

- **Dados de projeto auxiliado por computador (CAD)**, que inclui informações espaciais sobre como são construídos os objetos – como prédios, carros e aeronaves. Outros exemplos importantes de bancos de dados de projeto auxiliado por computador são layouts de circuito integrado e dispositivos eletrônicos.
- **Dados geográficos**, como mapas rodoviários, mapas de uso de terra, mapas de levantamento topográfico, mapas políticos mostrando fronteiras, mapas de propriedade de terra e assim por diante. **Sistemas de informações geográficas** são bancos de dados de uso especial moldados para armazenar dados geográficos.

O suporte para dados geográficos foi acrescentado a muitos sistemas de banco de dados, como o IBM DB2 Spatial Extender, o Informix Spatial Datablade e o Oracle Spatial.

Representação de informações geométricas

A Figura 24.2 ilustra como as diversas construções geométricas podem ser representadas em um banco de dados, em um padrão normalizado. Enfatizamos aqui que as informações geométricas podem ser representadas de várias maneiras diferentes, e descrevemos somente uma delas.

Um *segmento de linha* pode ser representado pelas coordenadas de suas extremidades. Por exemplo, em um banco de dados de mapa, as duas coordenadas de um ponto se-

1. Muitos pesquisadores de banco de dados temporal acham que esse tipo deveria ter sido chamado **span**, pois ele não especifica um horário inicial ou final exato, mas apenas o período de tempo entre os dois.

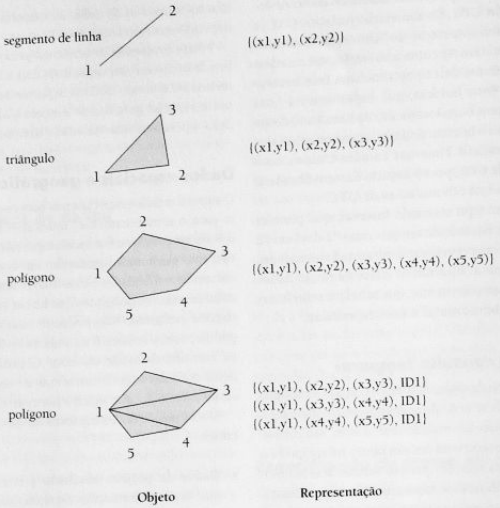


Figura 24.2 Representação de construções geométricas.

riam sua latitude e longitude. Uma *polilinha* (também chamada de *linestring*) consiste em uma seqüência conectada de segmentos de linha e pode ser representada por uma lista contendo as coordenadas das extremidades dos segmentos, em seqüência. Podemos representar aproximadamente uma curva arbitrária de polilinhas, particionando a curva em uma seqüência de segmentos. Essa representação é útil para recursos bidimensionais como estradas; aqui, a largura da estrada é tão pequena em relação ao tamanho do mapa inteiro que pode ser considerada bidimensional. Alguns sistemas também admitem *arcos circulares* como primitivas, permitindo que as curvas sejam representadas como seqüências de arcos.

Podemos representar um *polígono* listando seus vértices em ordem, como na Figura 24.2.[2] A lista de vértices especifica o limite de uma região poligonal. Em uma representação alternativa, um polígono pode ser dividido em um conjunto de triângulos, como mostra a Figura 24.2. Esse processo é chamado de **triangulação**, e qualquer polígono pode ser triangulado. O polígono complexo pode receber um identificador, e cada um dos triângulos em que ele é dividido carrega o identificador do polígono. Círculos e elipses podem ser representados pelos tipos correspondentes, ou podem ser aproximados por polígonos.

Representações de polilinhas ou polígonos baseadas em lista normalmente são convenientes para processamento de consulta. Essas representações não na primeira forma normal são usadas quando têm suporte do banco de dados básico. Para usar tuplas de tamanho fixo (na primeira forma normal) para representar polilinhas, podemos dar à polilinha ou curva um identificador, e podemos representar cada segmento como uma tupla separada, que também carrega consigo o identificador da polilinha ou curva. De modo semelhante, a representação triangulada de polígonos permite uma representação relacional da primeira forma normal dos polígonos.

A representação de pontos e segmentos de linha no espaço tridimensional é semelhante à sua representação no espaço bidimensional, sendo que a única diferença é que os pontos possuem um componente z extra. De modo semelhante, a representação de figuras planares – como triângu-

2. Algumas referências usam o termo *polígono fechado* para se referir ao que chamamos de polígonos, e se referem a polilinhas como polígonos abertos.

los, retângulos e outros polígonos – não muda muito quando passamos para três dimensões. Tetraedros e cubos podem ser representados da mesma maneira que triângulos e retângulos. Podemos representar poliedros quaisquer dividindo-os em tetraedros, assim como triangulamos os polígonos. Também podemos representá-los listando suas faces, cada qual sendo um polígono, junto com uma indicação de qual lado da face está dentro do poliedro.

Bancos de dados de projeto

Sistemas de projeto auxiliado por computador (CAD) tradicionalmente armazenavam dados na memória durante a edição ou outro processamento, e escreviam os dados de volta em um arquivo ao final de uma sessão de edição. As desvantagens desse esquema incluem o custo (complexidade de programação, bem como o custo do tempo) da transformação de dados de uma forma para outra, e a necessidade de ler um arquivo inteiro mesmo que somente algumas partes dele sejam necessárias. Para grandes projetos, como o projeto de um circuito integrado de grande escala de integração ou o projeto de uma aeronave inteira, pode ser impossível manter o projeto completo na memória. Os projetistas dos bancos de dados orientados a objeto foram motivados em grande parte pelos requisitos de banco de dados dos sistemas CAD. Os bancos de dados orientados a objeto representam componentes do projeto como objetos, e as conexões entre os objetos indicam como o projeto é estruturado.

Os objetos armazenados em um banco de dados de projeto geralmente são objetos geométricos. Objetos geométricos bidimensionais simples incluem pontos, linhas, triângulos, retângulos e, em geral, polígonos. Objetos bidimensionais complexos podem ser formados a partir de objetos simples por meio das operações de união, interseção e diferença. De modo semelhante, objetos tridimensionais complexos podem ser formados a partir de objetos mais simples, como esferas, cilindros e cubos, pelas operações de união, interseção e diferença, como na Figura 24.3. As superfícies tridimensionais também podem ser representadas por **modelos wireframe**, que basicamente modelam a superfície como um conjunto de objetos mais simples, como segmentos de linha, triângulos e retângulos.

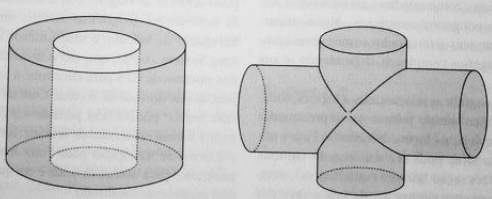
Os bancos de dados de projeto também armazenam informações não especiais sobre objetos, como o material a partir do qual os objetos são construídos. Normalmente, podemos modelar essas informações por técnicas padronizadas de modelagem de dados. Nos preocupamos aqui apenas com os aspectos espaciais.

Diversas operações espaciais precisam ser realizadas em um projeto. Por exemplo, o projetista pode querer obter a parte do projeto que corresponde a uma região de interesse específica. As estruturas de índice espacial, discutidas na seção "Indexação de dados espaciais", são úteis para essas tarefas. As estruturas de índice espacial são multidimensionais, lidando com dados bi e tridimensionais, em vez de lidar apenas com a ordenação unidimensional simples, fornecida pelas árvores B+.

As restrições de integridade espacial, como "dois tubos não podem estar no mesmo local", são importantes nos bancos de dados de projeto, para impedir erros de interferência. Esses erros normalmente ocorrem se o projeto for realizado manualmente, e só são detectados quando um protótipo está sendo construído. Como resultado, esses erros podem ser dispendiosos de se consertar. O suporte do banco de dados para as restrições de integridade espacial ajuda as pessoas a evitar erros de projeto, mantendo assim o projeto coerente. A implementação dessas verificações de integridade novamente depende da disponibilidade de estruturas de índice multidimensional eficientes.

Dados geográficos

Os dados geográficos são espaciais por natureza, mas diferem dos dados de projeto de certas maneiras. Mapas e ima-



(a) Diferença de cilindros (b) União de cilindros

Figura 24.3 Objetos tridimensionais complexos.

gens de satélite são exemplos típicos de dados geográficos. Os mapas podem oferecer não apenas informações de local – sobre fronteiras, rios e estradas, por exemplo – mas também informações muito mais detalhadas associadas a locais, como elevação, tipo de solo, uso de terra e índices pluviométricos anuais.

Os **dados geográficos** podem ser categorizados em dois tipos:

- **Dados de rastreio**. Esses dados consistem em mapas de bits ou mapas de pixels, em duas ou mais dimensões. Um exemplo típico de uma imagem de rastreio bidimensional é uma imagem de satélite da camada de nuvens, em que cada pixel armazena a visibilidade da nuvem em determinada área. Esses dados podem ser tridimensionais – por exemplo, a temperatura em diferentes altitudes em diferentes regiões, novamente medida com a ajuda de um satélite. O tempo poderia formar outra dimensão – por exemplo, as medidas de temperatura da superfície em diferentes pontos no tempo. Os bancos de dados de projeto geralmente não armazenam dados de rastreio.
- **Dados de vetor**. Dados de vetor são construídos a partir de objetos geométricos básicos, como pontos, segmentos de linha, triângulos e outros polígonos em duas dimensões, e cilindros, esferas, cubos e outros poliedros em três dimensões.

  Os dados de mapa normalmente são representados em formato de vetor. Rios e estradas podem ser representados como uniões de vários segmentos de linha. Estados e países podem ser representados como polígonos. Informações topológicas, como altura, podem ser representadas por uma superfície dividida em polígonos cobrindo regiões de mesma altura, com um valor de altura associado a cada polígono.

Representação de dados geográficos

Os recursos geográficos, como estados e grandes lagos, são representados como polígonos complexos. Alguns recursos, como rios, podem ser representados como curvas complexas ou como polígonos complexos, dependendo se sua largura é relevante.

As informações geográficas relacionadas a regiões, como o índice pluviométrico anual, podem ser representadas como um array – ou seja, na forma de rastreio. Para a eficiência do espaço, o array pode ser armazenado em uma forma compactada. Na seção "Árvores quadráticas", estudamos uma representação alternativa de tais arrays por uma estrutura de dados chamada *árvore quadrática*.

Conforme observamos na seção "Dados geográficos", podemos representar a informação de região em formato de vetor usando polígonos, em que cada polígono é uma região dentro da qual o valor do array é o mesmo. A representação de vetor é mais compacta do que a representação de rastreio em algumas aplicações. Ela também é mais precisa para algumas tarefas, como a representação de estradas, em que a divisão da região (que pode ser muito grande) em pixels leva a uma perda de precisão na informação de localização. Porém, a representação de vetor é inadequada para aplicações em que os dados são intrinsecamente baseados em rastreio, como imagens de satélite.

Aplicações de dados geográficos

Os bancos de dados geográficos possuem diversos usos, incluindo os serviços de mapa on-line; sistemas de navegação de veículo; informações de rede de distribuição para serviços públicos, como sistemas de telefone, eletricidade e água; e informações de uso de terra para ecologistas e planejadores.

Os serviços de mapa rodoviário formam uma aplicação muito utilizada de dados de mapa. No nível mais simples, esses sistemas podem ser usados para gerar mapas de estrada on-line de uma região desejada. Um benefício importante dos mapas on-line é que é fácil escalar os mapas ao tamanho desejado – ou seja, aproximar ou recuar para localizar recursos relevantes. Os serviços de mapa rodoviário também armazenam informações sobre estradas e serviços, como o layout de estradas, limites de velocidade nas estradas, condições da estrada, conexões entre estradas e restrições de mão única. Com essa informação adicional sobre as estradas, os mapas podem ser usados de modo a obter as direções para ir de um lugar a outro e para o planejamento automático de viagem. Os usuários podem consultar informações on-line sobre serviços para localizar, por exemplo, hotéis, postos de combustíveis ou restaurantes com cardápios e preços desejados.

Os sistemas de navegação de veículo são montados em automóveis, que oferecem serviços de mapas rodoviários e planejamento de viagem. Um acréscimo útil a um sistema de informações geográficas móveis, como um sistema de navegação de veículo, é uma unidade de **Global Positioning System** (**GPS**), que usa o broadcast de informações dos satélites de GPS para encontrar o local atual com uma precisão de dezenas de metros. Com tal sistema, um motorista nunca[3] poderá ficar perdido – a unidade de GPS encontra o local em termos de latitude, longitude e elevação e o sistema de navegação pode consultar o banco de dados geográfico para descobrir onde e em que estrada o veículo está atualmente localizado.

3. Bem, muito dificilmente!

Os bancos de dados geográficos para informações de utilidade pública estão se tornando cada vez mais importantes à medida que a rede de cabos e tubos enterrados cresce. Sem mapas detalhados, o trabalho executado por uma companhia pode danificar os cabos de outra companhia, resultando em interrupção de serviço em grande escala. Os bancos de dados geográficos, junto com sistemas precisos de descoberta de local, podem ajudar a evitar tais problemas.

Até aqui, explicamos por que os bancos de dados espaciais são úteis. No restante desta seção, estudaremos detalhes técnicos, como a representação e a indexação de informações espaciais.

Consultas espaciais

Existem diversos tipos de consultas que envolvem locais espaciais.

- Consultas de **proximidade** solicitam objetos que se encontram próximos de um local especificado. Uma consulta para encontrar todos os restaurantes que estão dentro de determinada distância de determinado ponto é um exemplo de uma consulta de proximidade. A **consulta do vizinho mais próximo** solicita o objeto que está mais próximo de um ponto especificado. Por exemplo, podemos querer encontrar o posto de combustível mais próximo. Observe que essa consulta não precisa especificar um limite na distância, e por isso podemos pedi-la até mesmo se não tivermos idéia da distância a que se encontra o posto de combustíveis.
- **Consultas por região** tratam de regiões espaciais. Essa consulta pode pedir objetos que se encontram parcial ou totalmente dentro de uma região especificada. Um exemplo é uma consulta para encontrar todas as revendas dentro dos limites geográficos de determinada cidade.
- As consultas também podem solicitar **interseções** e **uniões** de regiões. Por exemplo, dada a informação da região, como o índice pluviométrico anual e a densidade populacional, uma consulta pode solicitar todas as regiões com um índice pluviométrico anual baixo e uma densidade populacional alta.

As consultas que calculam interseções de regiões podem ser imaginadas como calculando a **junção espacial** de duas relações espaciais – por exemplo, uma representando o índice pluviométrico e a outra representando a densidade populacional – com o local desempenhando o papel de atributo de junção. Em geral, dadas duas relações, cada uma contendo objetos espaciais, a junção espacial das duas relações gera pares de objetos com interseção ou as regiões de interseção de tais pares.

Vários algoritmos de junção calculam de forma eficiente as junções espaciais sobre dados de vetor. Embora a junção de loop aninhado (com índices espaciais) possa ser usada, as junções de hash e as junções de sort-merge não podem ser usadas sobre dados espaciais. Os pesquisadores propuseram técnicas de junção baseadas na travessia coordenada de estruturas de índice espaciais sobre as duas relações. Veja as notas bibliográficas para obter mais informações. Em geral, as consultas sobre dados espaciais podem ter uma combinação de requisitos espaciais e não espaciais. Por exemplo, podemos querer encontrar o restaurante mais próximo que tenha cardápios vegetarianos, e que cobre menos de $10 por uma refeição.

Como os dados espaciais são inerentemente gráficos, normalmente os consultamos usando uma linguagem de consulta gráfica. Os resultados dessas consultas também são exibidos graficamente, em vez de em tabelas. O usuário pode invocar diversas operações na interface, como a escolha de uma área a ser vista (por exemplo, apontando e clicando em subúrbios a oeste de Manhattan), recuo e aproximação, escolha do que apresentar com base em condições de seleção (por exemplo, casas com mais de três quartos), camada de múltiplos mapas (por exemplo, casas com mais de três quartos sobrepostas a um mapa mostrando áreas com baixas taxas de criminalidade), e assim por diante. A interface gráfica constitui o front-end. As extensões da SQL foram propostas para permitir que os bancos de dados relacionais armazenem e recuperem informações espaciais de forma eficiente, e também permitem que as consultas misturem condições espaciais e não espaciais. As extensões incluem permitir tipos de dados abstratos, como linhas, polígonos e mapas de bits e condições espaciais, como *contains* ou *overlaps*.

Indexação de dados espaciais

Os índices são necessários para o acesso eficiente aos dados espaciais. As estruturas de índice tradicionais, como índices de hash e árvores B, não são adequadas, pois lidam apenas com dados unidimensionais, enquanto os dados espaciais normalmente são de duas ou mais dimensões.

Árvores k-d

Para entender como indexar dados espaciais consistindo em duas ou mais dimensões, consideramos primeiro a indexação de pontos em dados unidimensionais. Três estruturas, como árvores binárias e árvores B, operam dividindo o espaço sucessivamente em partes menores. Por exemplo, cada nó interno de uma árvore binária divide um intervalo unidimensional em dois. Os pontos que se encontram na parte esquerda vão para a subárvore esquerda; os pontos

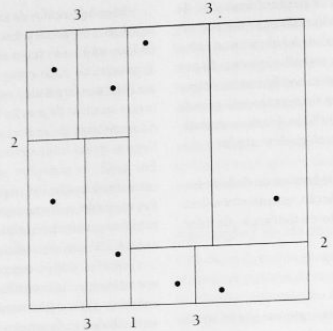


Figura 24.4 Divisão de espaço por uma árvore k-d.

que se encontram na partição direita vão para a subárvore direita. Em uma árvore binária balanceada, a partição é escolhida de modo que aproximadamente metade dos pontos armazenados na subárvore caia em cada partição. De modo semelhante, cada nível de uma árvore B divide um intervalo unidimensional em várias partes.

Podemos usar essa intuição para criar estruturas de árvore para o espaço bidimensional, além de espaços de maiores dimensões. Uma estrutura de árvore chamada **árvore k-d** foi uma das primeiras estruturas usadas para a indexação em várias dimensões. Cada nível de uma árvore k-d divide o espaço em dois. O particionamento é feito ao longo de uma dimensão no nó no nível superior da árvore, junto com outra dimensão nos nós no próximo nível, e assim por diante, percorrendo as dimensões. O particionamento prossegue de modo que, em cada nó, aproximadamente metade dos pontos armazenados na subárvore caia em um lado e metade caia no outro. O particionamento para quando um nó tem menos do que determinado número máximo de pontos. A Figura 24.4 mostra um conjunto de pontos no espaço bidimensional e uma representação de árvore k-d do conjunto de pontos. Cada linha corresponde a um nó na árvore, e o número máximo de pontos em um nó de folha foi definido como 1. Cada linha da figura (fora a caixa externa) corresponde a um nó na árvore k-d. A numeração das linhas na figura indica o nível da árvore em que o nó correspondente aparece.

A **árvore k-d-B** estende a árvore k-d para permitir vários nós filhos para cada nó interno, assim como uma árvore B estende uma árvore binária, para reduzir a altura da árvore. As árvores k-d-B são mais apropriadas para o armazenamento secundário do que as árvores k-d.

Árvores quadráticas

Uma representação alternativa para os dados bidimensionais é uma **árvore quadrática**. Um exemplo de divisão de espaço por uma árvore quadrática aparece na Figura 24.5. O conjunto de pontos é o mesmo daquele da Figura 24.4. Cada nó de uma árvore quadrática está associado a uma região retangular do espaço. O nó superior está associado ao espaço de destino inteiro. Cada nó não-folha em uma árvore quadrática divide sua região em quatro quadrantes de mesmo tamanho, e, de modo semelhante, cada nó possui quatro nós filhos, correspondentes aos quatro quadrantes. Os nós de folha têm entre zero e algum número máximo fixo de pontos. De modo semelhante, se a região correspondente a um nó tiver mais do que o número máximo de pontos, os nós filhos são criados para esse nó. No exemplo da Figura 24.5, o número máximo de pontos em um nó de folha é definido como 1.

Esse tipo de árvore quadrática é chamado de **árvore quadrática PR**, para indicar que ela armazena pontos e que a divisão do espaço é feita com base em regiões, em vez do conjunto real de pontos armazenados. Podemos usar **árvores quadráticas de região** para armazenar informações de array (rastreio). Um nó em uma árvore quadrática de região é um nó de folha se todos os valores de array na região que ele cobre forem iguais. Caso contrário, ele é subdividido ainda mais em quatro filhos de mesma área, e, portanto, é um nó interno. Cada nó na árvore quadrática de região corresponde a um subarray de valores. Os subarrays correspondentes às folhas ou contêm apenas um único elemento de array ou possuem vários elementos de array, todos tendo o mesmo valor.

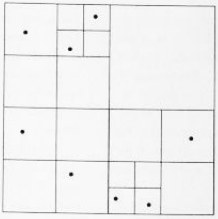


Figura 24.5 Divisão de espaço por uma árvore quadrática.

A indexação dos segmentos de linha e polígonos apresenta novos problemas. Existem extensões de árvores k-d e árvores quadráticas para essa tarefa. Porém, um segmento de linha ou polígono pode cruzar uma linha de particionamento. Se o fizer, ele precisa ser dividido e representado em cada uma das subárvores em que suas partes ocorrem. Várias ocorrências de um segmento de linha ou polígono podem resultar em ineficiências no armazenamento, além de ineficiências na consulta.

Árvores R

Uma estrutura de armazenamento chamada **árvore-R** é útil para a indexação de objetos como pontos, segmentos de linha, retângulos e outros polígonos. Uma árvore-R é uma estrutura de árvore balanceada com os objetos indexados armazenados nos nós de folha, semelhante à árvore B^+. Porém, em vez de um intervalo de valores, uma **caixa de contorno** retangular é associada a cada nó da árvore. A caixa de contorno de um nó de folha é o menor retângulo paralelo aos eixos que contém todos os objetos armazenados no nó de folha. A caixa de contorno dos nós internos, de modo semelhante, é o menor retângulo paralelo aos eixos que contém as caixas de contorno dos seus nós filhos. A caixa de contorno de um objeto (como um polígono) é definida, de modo semelhante, como o menor retângulo, paralelo aos eixos, que contém o objeto.

Cada nó interno armazena as caixas de contorno dos nós filhos junto com os ponteiros para os nós filhos. Cada nó de folha armazena os objetos indexados, e opcionalmente pode armazenar as caixas de contorno dos objetos; as caixas de contorno ajudam a agilizar verificações de sobreposições do retângulo com os objetos indexados – se um retângulo de consulta não se sobrepuser à caixa de contorno de um objeto, ele também não poderá se sobrepor ao objeto. (Se os objetos indexados forem retângulos, naturalmente não será preciso armazenar caixas de contorno, pois elas são idênticas aos retângulos.)

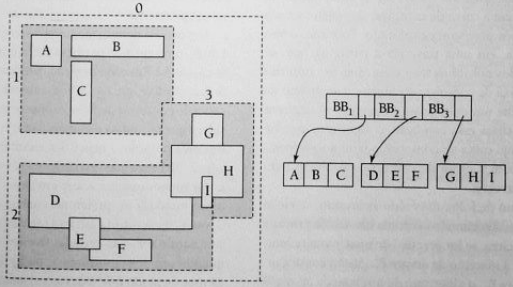


Figura 24.6 Uma árvore R.

A Figura 24.6 mostra um exemplo de um conjunto de retângulos (desenhados com uma linha sólida) e as caixas de contorno (desenhadas com uma linha tracejada) dos nós da árvore R para o conjunto de retângulos. Observe que as caixas de contorno aparecem com espaço extra dentro delas, para que se destaquem no desenho. Na realidade, as caixas seriam menores e ficariam coladas aos objetos que elas contêm; ou seja, cada lado de uma caixa de contorno B tocaria pelo menos em um dos objetos ou caixas de contorno que estão contidas em B.

A própria árvore-R está no lado direito da Figura 24.6. A figura refere-se às coordenadas da caixa de contorno i como BB_i.

Agora, vejamos como implementar as operações de busca, inserção e exclusão em uma árvore R.

- **Busca**. Como mostra a figura, as caixas de contorno associadas a nós irmãos podem se sobrepor; nas árvores B+, árvores k-d e árvores quadráticas, ao contrário, os intervalos não se sobrepõem. Uma busca por objetos contendo um ponto, portanto, precisa seguir *todos* os nós filhos cujas caixas de contorno associadas contêm o ponto; como resultado, vários caminhos podem ter de ser pesquisados. De modo semelhante, uma consulta para encontrar todos os objetos que cruzam determinado objeto precisa descer por nó em que o retângulo associado cruza o objeto indicado.
- **Inserção**. Quando inserimos um objeto em uma árvore R, selecionamos um nó de folha para manter o objeto. O ideal é escolher um nó de folha que tenha espaço para manter uma nova entrada, cuja caixa de contorno contenha a caixa de contorno do objeto. Porém, esse nó pode não existir; mesmo que existisse, encontrar o nó pode ser muito dispendioso, pois não é possível encontrá-lo por uma única travessia a partir da raiz. Em cada nó interno, podemos encontrar vários filhos cujas caixas de contorno contêm a caixa de contorno do objeto, e cada um desses filhos precisa ser explorado. Portanto, como uma heurística, em uma travessia a partir da raiz, se qualquer um dos nós filhos tiver uma caixa de contorno contendo a caixa de contorno do objeto, o algoritmo da árvore R escolhe um deles arbitrariamente. Se nenhum dos filhos satisfizer essa condição, o algoritmo escolhe um nó filho cuja caixa de contorno possui a sobreposição máxima com a caixa de contorno do objeto para continuar a travessia.

  Quando o nó de folha tiver sido alcançado, se ele já estiver cheio, o algoritmo realiza sua divisão (e propaga a divisão para cima, se for preciso) de uma maneira muito semelhante à inserção da árvore B+. Assim como a inserção da árvore B+, o algoritmo de inserção da árvore R garante que a árvore permaneça balanceada. Além do mais, isso garante que as caixas de contorno dos nós de folha, juntamente com os nós internos, permaneçam consistentes; ou seja, as caixas de contorno das folhas contêm todas as caixas de contorno dos objetos armazenados na folha, enquanto as caixas de contorno para os nós internos contêm todas as caixas de contorno dos nós filhos.

  A diferença principal do procedimento de inserção da árvore B+ está no modo como o nó é dividido. Em uma árvore B+, é possível encontrar um valor tal que metade das entradas seja menor que o ponto intermediário e metade seja maior que o valor. Essa propriedade não generaliza além de uma dimensão; ou seja, para mais de uma dimensão, nem sempre é possível dividir as entradas em dois conjuntos para que suas caixas de contorno não se sobreponham. Em vez disso, como uma heurística, o conjunto de entradas S pode ser dividido em dois conjuntos disjuntos S_1 e S_2, de modo que as caixas de contorno de S_1 e S_2 tenham a área total mínima; outra heurística seria dividir as entradas em dois conjuntos S_1 e S_2 de modo que S_1 e S_2 tenham sobreposição mínima. Os dois nós resultantes da divisão teriam as entradas em S_1 e S_2, respectivamente. O custo para localizar as divisões com o mínimo de área total ou sobreposição pode ser muito grande, de modo que heurísticas menos dispendiosas, como a heurística da *divisão quadrática*, são utilizadas. (A heurística recebe seu nome do fato de que ela exige um tempo que é o quadrado do número de entradas.)

  A heurística da **divisão quadrática** funciona desta maneira: primeiro, ela seleciona um par de entradas a e b a partir de S, de modo que colocá-los no mesmo nó resultaria em uma caixa de contorno com o máximo de espaço desperdiçado; ou seja, a área da caixa de contorno mínima de a e b menos a soma das áreas de a e b é a maior. A heurística coloca as entradas a e b nos conjuntos S_1 e S_2, respectivamente.

  Depois, ela acrescenta iterativamente as entradas restantes, uma entrada por iteração, a um dos dois conjuntos S_1 ou S_2. Em cada iteração, para cada entrada restante e, considere que $i_{e,1}$ indica o aumento no tamanho da caixa de contorno de S_1 se e for acrescentado a S_1 e considere que $i_{e,2}$ indica o aumento correspondente para S_2. Em cada iteração, a heurística escolhe uma das entradas com a diferença máxima de $i_{e,1}$ e $i_{e,2}$ e a acrescenta a S_1 se $i_{e,1}$ for menor que $i_{e,2}$, e a S_2 em caso contrário. Ou seja, uma entrada com "preferência máxima" para um dentre S_1 ou S_2 é escolhida em cada iteração. A iteração pára quando todas as entradas tiverem sido atribuídas, ou quando um dos conjuntos S_1 ou S_2 tiver entradas suficientes de modo que todas as entradas restantes tenham de ser acrescentadas ao outro conjunto para que os nós

construídos de S_1 e S_2 tenham o mínimo necessário de ocupação. A heurística, então, acrescenta todas as entradas não atribuídas ao conjunto com menos entradas.

- Exclusão. A exclusão pode ser realizada como uma exclusão de árvore B+, apanhando emprestadas as entradas de nós irmãos, ou mesclando os nós irmãos se um nó não estiver cheio. Uma técnica alternativa redistribui todas as entradas dos nós não cheios para nós irmãos, com o objetivo de melhorar o agrupamento de entradas na árvore R.

Consulte as referências bibliográficas para obter mais detalhes sobre operações de inserção e exclusão em árvores R, além das variantes de árvores R, chamadas árvores R* e árvores R+.

A eficiência de armazenamento das árvores R é melhor do que a das árvores k-d ou árvores quadráticas, pois um objeto só é armazenado uma vez, e podemos garantir facilmente que cada nó esteja pelo menos completo até a metade. Porém, a consulta pode ser mais lenta, pois vários caminhos precisam ser pesquisados. As junções espaciais são mais simples com árvores quadráticas do que com árvores R, pois todas as árvores quadráticas em uma região são divididas da mesma maneira. Porém, devido à sua melhor eficiência de armazenamento, e sua semelhança com árvores B, as árvores R e suas variantes provaram ser populares nos sistemas de banco de dados que admitem dados espaciais.

Bancos de dados de multimídia

Os dados de multimídia, como imagens, áudio e vídeo – uma forma cada vez mais popular de dados – estão hoje quase sempre armazenados fora do banco de dados, nos sistemas de arquivo. Esse tipo de armazenamento não é um problema quando o número de objetos de multimídia é relativamente pequeno, pois os recursos fornecidos pelos bancos de dados normalmente não são importantes.

Porém, os recursos de banco de dados se tornam importantes quando o número de objetos de multimídia armazenados é grande. Questões como atualizações transacionais, facilidades de consulta e indexação se tornam importantes. Os objetos de multimídia normalmente possuem atributos descritivos, como aqueles indicando quando eles foram criados, quem os criou e a que categoria eles pertencem. Uma técnica para a criação de um banco de dados para esses objetos de multimídia é usar bancos de dados a fim de armazenar os atributos descritivos e registrar os arquivos em que os objetos de multimídia estão armazenados.

Porém, armazenar multimídia fora do banco de dados torna mais difícil oferecer funcionalidade de banco de dados, como indexação com base no conteúdo de dados de multimídia real. Isso também pode levar a inconsistências, como um arquivo que deve ser anotado no banco de dados, mas cujo conteúdo está faltando, ou vice-versa. Portanto, é desejável armazenar os próprios dados no banco de dados.

Várias questões precisam ser resolvidas se os dados de multimídia tiverem de ser armazenados em um banco de dados.

- O banco de dados precisa dar suporte a objetos grandes, pois os dados de multimídia como vídeos podem ocupar até alguns gigabytes de armazenamento. Muitos sistemas de banco de dados não admitem objetos maiores que alguns gigabytes. Objetos maiores poderiam ser divididos em partes menores e armazenados no banco de dados. Como alternativa, o objeto de multimídia pode ser armazenado em um sistema de arquivos, mas o banco de dados pode conter um ponteiro para o objeto; o ponteiro normalmente seria um nome de arquivo. O padrão SQL/MED (MED significa Management of External Data) permite que dados externos, como arquivos, sejam tratados como se fizessem parte do banco de dados. Com SQL/MED, o objeto pareceria ser parte do banco de dados, mas pode ser armazenado externamente.

  Discutimos os formatos de dados de multimídia na próxima seção.
- A recuperação de alguns tipos de dados, como áudio e vídeo, tem o requisito de que a remessa dos dados precisa prosseguir em um ritmo constante garantido. Esses dados às vezes são chamados de **dados isócronos**, ou **dados de mídia contínua**. Por exemplo, se os dados de áudio não forem fornecidos em tempo, haverá lacunas no som. Se os dados forem fornecidos muito rapidamente, os buffers do sistema poderão estourar, resultando em perda de dados. Discutimos os dados de mídia contínua na seção "Dados de mídia contínua".
- A recuperação baseada em semelhança é necessária em muitas aplicações de banco de dados. Por exemplo, em um banco de dados que armazena imagens de impressão digital, uma imagem da impressão digital a consultar é fornecida, e as impressões digitais no banco de dados que são semelhantes à imagem da consulta precisam ser apanhadas. Estruturas de índice como árvores B+ e árvores R não podem ser usadas para essa finalidade; estruturas de índice especiais precisam ser criadas. Discutimos a recuperação baseada em semelhança na seção "Recuperação baseada em semelhança".

Formatos de dados de multimídia

Devido ao grande número de bytes exigidos para representar dados de multimídia, é essencial que os dados de multimídia sejam armazenados e transmitidos em formato compactado. Para dados de imagem, o formato mais usado é

JPEG, que possui o nome da agência de padrões que o criou, o *Joint Picture Experts Group*. Podemos armazenar dados de vídeo codificando cada quadro de vídeo em formato JPEG, mas essa codificação é desperdiçadora, pois os quadros sucessivos de um vídeo normalmente são quase iguais. O *Moving Picture Experts Group* desenvolveu a série *MPEG* de padrões para codificar dados de vídeo e áudio; essas codificações exploram semelhanças entre uma seqüência de quadros para alcançar um maior grau de compactação. O padrão *MPEG-1* armazena um minuto de vídeo e áudio a 30 quadros por segundo em aproximadamente 12,5 megabytes (em comparação aos quase 75 megabytes para o vídeo apenas em JPEG). Porém, a codificação MPEG-1 introduz alguma perda de qualidade de vídeo, a um nível aproximadamente comparável ao da fita de vídeo VHS. O padrão *MPEG-2* foi criado para sistemas de broadcast digital e discos de vídeo digital (DVDs); ele introduz apenas uma perda desconsiderável na qualidade do vídeo. MPEG-2 compacta 1 minuto de vídeo e áudio em aproximadamente 17 megabytes. MPEG-4 oferece técnicas para compactar vídeo ainda mais, com largura de banda variável para dar suporte à remessa de dados de vídeo por redes com uma grande gama de larguras de banda. Vários padrões concorrentes são usados para a codificação de áudio, incluindo *MP3*, que significa MPEG-1 Layer 3, RealAudio, Windows Media Audio e outros formatos.

Dados de mídia contínua

Os tipos mais importantes de dados de mídia contínua são dados de vídeo e áudio (por exemplo, um banco de dados de filmes). Os sistemas de mídia contínua são caracterizados por seus requisitos de remessa de informações em tempo real:

- Os dados precisam ser remetidos de modo suficiente rápido para que não haja lacunas no resultado em vídeo ou áudio.
- Os dados precisam ser entregues em uma velocidade que não cause estouro dos buffers do sistema.
- O sincronismo entre fluxos de dados distintos precisa ser mantido. Essa necessidade surge, por exemplo, quando o vídeo de uma pessoa falando precisa mostrar os lábios se movendo de forma sincronizada com o áudio da pessoa falando.

Para fornecer dados de modo previsível na hora certa a um grande número de consumidores dos dados, a busca de dados do disco precisa ser cuidadosamente coordenada. Normalmente, os dados são apanhados em ciclos periódicos. Em cada ciclo, digamos, de n segundos, n segundos de dados são apanhados para cada cliente e armazenados em buffers da memória, enquanto os dados apanhados no ciclo anterior estão sendo enviados aos consumidores a partir dos buffers de memória. O período do ciclo é um meio-termo: um período curto usa menos memória, mas exige mais movimento de braço do disco, que é um desperdício de recursos, enquanto um período longo reduz o movimento do braço do disco, mas aumenta os requisitos de memória e pode adiar a remessa inicial dos dados. Quando uma nova solicitação chega, o **controle de admissão** entra em ação: ou seja, o sistema verifica se a solicitação pode ser satisfeita com recursos disponíveis (em cada período); nesse caso, ela é admitida; caso contrário, é rejeitada.

Uma extensa pesquisa sobre a remessa de dados de mídia contínua lidou com questões como o manuseio de arrays de discos e o tratamento de falha do disco. Veja os detalhes nas referências bibliográficas.

Vários fornecedores oferecem servidores de vídeo por demanda. Os sistemas atuais são baseados em sistemas de arquivos, pois os sistemas de banco de dados existentes não oferecem a resposta em tempo real que essas aplicações precisam. A arquitetura básica de um sistema de vídeo por demanda compreende:

- **Servidor de vídeo.** Os dados de multimídia são armazenados em vários discos (normalmente, em uma configuração RAID). Os sistemas contendo um grande volume de dados podem usar o armazenamento terciário para dados acessados com menos freqüência.
- **Terminais.** As pessoas vêem dados de multimídia por meio de diversos dispositivos, coletivamente chamados de terminais. Alguns exemplos são computadores pessoais e televisões conectadas a um pequeno e barato computador, chamado **set-up box**.
- **Rede.** A transmissão de dados de multimídia de um servidor para vários terminais exige uma rede de alta capacidade.

O serviço de vídeo por demanda via redes a cabo está disponível em muitos lugares hoje, e por fim se tornará onipresente, assim como acontece hoje com a televisão a cabo e por radiodifusão.

Recuperação baseada em semelhança

Em muitas aplicações de multimídia, os dados são descritos apenas aproximadamente nesse banco de dados. Um exemplo são os dados de impressão digital da seção "Bancos de dados de multimídia". Outros exemplos são:

- **Dados de imagem.** Duas figuras ou imagens que são ligeiramente diferentes, conforme representadas no banco de dados, podem ser consideradas iguais por um

usuário. Por exemplo, um banco de dados pode armazenar projetos de marca comercial. Quando uma nova marca comercial deve ser registrada, o sistema pode precisar primeiro identificar todas as marcas comerciais semelhantes que foram registradas anteriormente.

- **Dados de áudio.** Interfaces de usuário baseadas em fala estão sendo desenvolvidas para permitir que o usuário dê um comando ou identifique um item de dados pela fala. A entrada do usuário precisa então ser testada por semelhança com aqueles comandos ou itens de dados armazenados no sistema.
- **Dados escritos à mão.** A entrada escrita à mão pode ser usada para identificar um item de dados ou comando escrito à mão, armazenado no banco de dados. Aqui, novamente, o teste de semelhança é exigido.

A noção de semelhança normalmente é subjetiva e específica do usuário. Porém, o teste de semelhança normalmente é mais bem-sucedido do que o reconhecimento de voz ou escrita manual, pois a entrada pode ser comparada com dados já no sistema e, assim, o conjunto de opções disponíveis ao sistema é limitado.

Existem vários algoritmos para encontrar as melhores combinações de determinada entrada pelo teste de semelhança. Alguns sistemas, incluindo a discagem por nome, sistema de telefone ativado por voz, foram implantados comercialmente. Consulte as referências nas notas bibliográficas.

Mobilidade e bancos de dados pessoais

Bancos de dados comerciais, em grande escala, tradicionalmente têm sido armazenados em instalações de computação centrais. Em aplicações de banco de dados distribuídas, normalmente tem havido forte administração central de banco de dados e rede. Duas tendências tecnológicas se combinaram para criar aplicações em que essa suposição de controle e administração central não é inteiramente correta:

1. O uso cada vez mais divulgado de computadores pessoais e, mais importante, de computadores de laptop ou notebook.
2. O desenvolvimento de uma infra-estrutura de comunicação digital sem fio relativamente barata, com base nas redes locais sem fio, redes celulares de pacotes digitais e outras tecnologias.

A **computação móvel** provou ser útil em muitas aplicações. Muitos viajantes a negócio utilizam computadores de laptop para que possam trabalhar e acessar dados em viagem. Os serviços de entrega utilizam computadores móveis para auxiliar no rastreamento de encomendas. Os serviços de resposta a emergência utilizam computadores móveis na cena dos desastres, emergências médicas e coisas desse tipo para acessar informações e entrar com dados pertencentes à situação. Os telefones celulares estão cada vez mais se tornando dispositivos que oferecem não apenas serviços de telefone, mas também computadores móveis, permitindo acesso por e-mail e pela Web. Novas aplicações de computadores móveis continuam a surgir.

A computação sem fio cria uma situação em que as máquinas não possuem mais locais e endereços de rede fixos. **Consultas dependentes de local** são uma classe interessante de consultas, que são motivadas por computadores móveis; nessas consultas, o local do usuário (comunicação) é um parâmetro da consulta. O valor do parâmetro de local é fornecido por um usuário ou, cada vez mais, por um sistema de posicionamento global (GPS). Um exemplo é um sistema de informações de um viajante, que oferece dados sobre hotéis, serviços de beira de estrada e coisas desse tipo para os motoristas. O processamento de consultas sobre serviços que estão adiante na rota atual precisa ser baseado no conhecimento do local do usuário, direção do movimento e velocidade. Cada vez mais, recursos de navegação estão sendo oferecidos como um recurso embutido nos automóveis.

A energia (alimentação por bateria) é um recurso escasso para a maioria dos computadores móveis. Essa limitação influencia muitos aspectos do projeto do sistema. Entre as consequências mais interessantes da necessidade de eficiência de energia estão os dispositivos móveis pequenos gastarem a maior parte do seu tempo dormindo, acordando por uma fração de segundo a cada segundo, ou mais, para verificar os dados que chegam e enviar os dados que saem. Esse comportamento possui um impacto significativo sobre os protocolos utilizados para a comunicação com dispositivos móveis. O uso de broadcasts de dados agendados para reduzir a necessidade de sistemas móveis transmitirem consultas é outra maneira de reduzir os requisitos de energia.

Quantidades cada vez maiores de dados podem residir em máquinas administradas por usuários, em vez de administradores de banco de dados. Além do mais, essas máquinas podem, às vezes, estar desconectadas da rede. Em muitos casos, existe um conflito entre a necessidade de o usuário continuar a trabalhar enquanto desconectado e a necessidade de consistência de dados global.

Nas seções "Um modelo de computação móvel" a "Desconectividade e consistência", discutimos as técnicas em uso e em desenvolvimento para lidar com os problemas de mobilidade e computação pessoal.

Um modelo de computação móvel

O ambiente de computação móvel consiste em computadores móveis, conhecidos como **hosts móveis**, e uma rede de

computadores interligados. Os hosts móveis se comunicam com a rede interligada por meio de computadores conhecidos como **estações de suporte móvel**. Cada estação de suporte móvel gerencia os hosts móveis dentro de sua **célula** – ou seja, a área geográfica que ela abrange. Os hosts móveis podem se mover entre as células, necessitando assim de um **repasse** de controle de uma estação de suporte móvel para outra. Como os hosts móveis podem, às vezes, ser desligados, um host pode deixar uma célula e rematerializar mais tarde em alguma célula distante. Portanto, os movimentos entre as células não são necessariamente entre células adjacentes. Dentro de uma área pequena, como um prédio, os hosts móveis podem estar conectados por uma rede local sem fio (LAN) que oferece conectividade de menor custo que uma rede celular remota, e isso reduz a sobrecarga dos repasses.

É possível que os hosts móveis se comuniquem diretamente sem a intervenção de uma estação de suporte móvel. Porém, essa comunicação só pode ocorrer entre hosts próximos. As formas diretas de comunicação estão se tornando mais prevalentes com o surgimento do padrão **Bluetooth**. Bluetooth usa rádio digital de curto alcance para permitir a conectividade sem fio dentro do intervalo de 10 metros em alta velocidade (até 721 kilobits por segundo). Inicialmente concebido como um substituto para os cabos, a maior promessa do Bluetooth está na conexão ocasional fácil de comunicações móveis, PDAs, telefones móveis e os chamados utensílios inteligentes.

Os sistemas de rede local sem fio baseados nos padrões 801.11 (a/b/g) são muito usados hoje, e os sistemas baseados no 802.16 (Wi-Max) começaram a aparecer em 2005.

A infra-estrutura de rede para computação móvel consiste, em grande parte, em duas tecnologias: redes locais sem fio e redes de telefonia celular baseadas em pacotes. Os primeiros sistemas celulares usavam tecnologia analógica e foram projetados para a comunicação por voz. Os sistemas digitais de segunda geração retiveram o foco nas aplicações de voz. Os sistemas de terceira geração (3G) e os chamados sistemas 2.5G utilizam redes baseadas em pacote e são mais adequados para aplicações de dados. Nessas redes, a voz é apenas uma das muitas aplicações (apesar de ser economicamente importante).

Bluetooth, LANs sem fio e redes de celular 2.5G e 3G tornam possível que uma grande variedade de dispositivos se comuniquem com baixo custo. Embora essa própria comunicação não pertença ao domínio de uma aplicação de banco de dados normal, a contabilidade, a monitoração e o gerenciamento de dados pertencentes a essa comunicação gerará bancos de dados imensos. A proximidade da comunicação sem fio gera uma necessidade de acesso em tempo real a muitos desses bancos de dados. Essa necessidade de acesso em tempo acrescenta outra dimensão às restrições sobre o sistema – uma questão que discutiremos melhor na seção "Bancos de dados na memória principal" do Capítulo 25.

As limitações de tamanho e potência de muitos computadores móveis levaram a hierarquias de memória alternativas. Em vez de (ou além de) armazenamento de disco, a memória flash, que discutimos na seção "Visão geral do meio de armazenamento físico" do Capítulo 11, pode ser incluída. Se o host móvel incluir um disco rígido, o disco pode ter permissão para girar mais lentamente quando não estiver em uso, para economizar energia. As mesmas considerações de tamanho e energia limitam o tipo e o tamanho da tela usada em um dispositivo móvel. Os projetistas de dispositivos móveis normalmente criam interfaces com o usuário de uso especial para trabalhar dentro dessas restrições. Porém, a necessidade de apresentar dados baseados na Web demandou a criação de padrões de apresentação. O **Wireless Application Protocol** (WAP) é um padrão para o acesso sem fio à Internet. Os navegadores baseados em WAP acessam páginas Web especiais, que utilizam a **linguagem de marcação sem fio** (WML – Wireless Markup Langauge), uma linguagem baseada em XML, criada para as restrições da navegação Web móvel e sem fio.

Roteamento e processamento de consulta

A rota entre um par de hosts pode mudar com o tempo se um dos dois hosts for móvel. Esse simples fato possui um efeito dramático no nível de rede, pois os endereços de rede baseados em local não são mais constantes dentro do sistema.

A mobilidade também afeta diretamente o processamento de consulta de banco de dados. Como vimos no Capítulo 22, temos de considerar os custos de comunicação quando escolhemos uma estratégia distribuída de processamento de consulta. A mobilidade resulta em mudar dinamicamente os custos de comunicação, complicando assim o processo de otimização. Além do mais, existem noções concorrentes de custo a serem consideradas:

- **Tempo do usuário** é um artigo altamente valioso em muitas aplicações de negócios.
- **Tempo de conexão** é a unidade pela qual os encargos monetários são atribuídos em alguns sistemas de celular.
- **Número de bytes ou pacotes transferidos** é a unidade pela qual os encargos são calculados em alguns sistemas de celular digitais.
- **Encargos baseados na hora do dia** variam, dependendo se a comunicação ocorre durante períodos de pico ou fora de pico.
- **Energia** é limitada. Normalmente, a alimentação da bateria é um recurso escasso, cujo uso precisa ser otimizado. Um princípio básico da comunicação por rádio é que é preciso menos energia para receber do que para trans-

mitir sinais de rádio. Assim, a transmissão e a recepção de dados impõem diferentes demandas de potência sobre o host móvel.

Dados de broadcast

Normalmente, é desejável que os dados solicitados com mais frequência sejam enviados em um ciclo contínuo pelas estações de suporte móveis, em vez de transmitidas aos hosts móveis por demanda. Uma aplicação típica de tais **dados de broadcast** são as informações de preço do mercado de ações. Existem dois motivos para se usar dados de broadcast. Primeiro, o host móvel evita o custo de energia para transmitir solicitações de dados. Segundo, os dados de broadcast podem ser recebidos por uma grande quantidade de hosts móveis ao mesmo tempo, sem custo extra. Assim, a largura de banda de transmissão disponível é utilizada de forma mais eficiente.

Um host móvel pode, então, receber dados à medida que são transmitidos, em vez de consumir energia transmitindo uma solicitação. O host móvel pode ter armazenamento não volátil local a sua disposição a fim de manter em cache os dados transmitidos, para possível uso posterior. Dada uma consulta, o host móvel pode otimizar custos de energia determinando se pode processar essa consulta apenas com dados em cache. Se os dados em cache forem insuficientes, haverá duas opções: esperar pelos dados a serem transmitidos ou transmitir uma solicitação para os dados. Para tomar essa decisão, o host móvel precisa saber quando os dados relevantes serão enviados.

Dados de broadcast podem ser transmitidos de acordo com um horário fixo ou variável. No primeiro caso, o host móvel utiliza o horário fixo conhecido para determinar quando os dados relevantes serão transmitidos. No segundo caso, o horário do broadcast precisa ser enviado em uma frequência de rádio bem conhecida e em intervalos de tempo bem conhecidos.

Com efeito, o meio de broadcast pode ser modelado como um disco com uma alta latência. As solicitações de dados podem ser imaginadas como sendo atendidas quando os dados solicitados forem enviados. Os horários de transmissão se comportam como índices no disco. As notas bibliográficas listam trabalhos de pesquisa recentes na área de gerenciamento de dados de broadcast.

Desconectividade e consistência

Como a comunicação sem fio pode ser paga com base no tempo de conexão, existe um incentivo para certos hosts móveis serem desconectados por períodos substanciais. Os computadores móveis sem conectividade sem fio estão desconectados na maior parte do tempo quando estão sendo usados, exceto periodicamente quando estão conectados aos seus computadores host, seja fisicamente ou por meio de uma rede de computadores.

Durante esses períodos de desconexão, o host móvel pode permanecer em operação. O usuário do host móvel pode emitir consultas e atualizações sobre os dados que residem ou que são colocados em cache localmente. Essa situação cria vários problemas, em particular:

- **Facilidade de recuperação:** as atualizações entradas em uma máquina desconectada podem ser perdidas se o host móvel experimentar uma falha catastrófica. Como o host móvel representa um único ponto de falha, o armazenamento estável não pode ser simulado bem.
- **Consistência:** os dados em cache local podem se tornar desatualizados, mas o host móvel não pode descobrir essa situação até que seja reconectado. De modo semelhante, as atualizações ocorrendo no host móvel não podem ser propagadas até que ocorra a reconexão.

Exploramos o problema de consistência no Capítulo 22, no qual discutimos o particionamento de rede, e mostramos pormenores disso aqui. Nos sistemas distribuídos com fio, o particionamento é considerado como um modo de falha; na computação móvel, o particionamento via desconexão faz parte do modo de operação normal. Portanto, é necessário permitir que o acesso aos dados prossiga apesar do particionamento, mesmo com o risco de alguma perda de consistência.

Para os dados atualizados apenas pelo host móvel, é simples propagar as atualizações quando o host móvel se reconectar. Porém, se o host móvel mantém em cache as cópias somente leitura dos dados que podem ser atualizados por outros computadores, os dados em cache podem se tornar inconsistentes. Quando o host móvel é conectado, ele pode receber relatórios de invalidação que o informam sobre entradas de cache desatualizadas. Porém, quando o host móvel é desconectado, ele pode perder um relatório de invalidação. Uma solução simples para esse problema é invalidar o cache inteiro na reconexão, mas uma solução extrema desse tipo é altamente dispendiosa. Vários esquemas de caching são citados nas notas bibliográficas.

Se as atualizações podem ocorrer no host móvel e em outro lugar, a detecção de atualizações em conflito é mais difícil. Os esquemas baseados em **número de versão** permitem as atualizações de arquivos compartilhados a partir de hosts desconectados. Esses esquemas não garantem que as atualizações serão consistentes. Em vez disso, elas garantem que, se dois hosts atualizarem independentemente a mesma versão de um documento, o conflito será finalmente detectado, quando os hosts trocarão informações diretamente ou por meio de um host comum.

O **esquema vetor de versão** detecta inconsistências quando as cópias de um documento forem atualizadas independentemente. Esse esquema permite que as cópias de um documento sejam armazenadas em vários hosts. Embora usemos o termo documento, o esquema pode ser aplicado a quaisquer outros itens de dados, como as tuplas de uma relação.

A idéia básica é que cada host i armazene, com sua cópia de cada documento d, um **vetor de versão** – ou seja, um conjunto de números de versão $\{V_{d,i}[j]\}$, com uma entrada para cada outro host j no qual o documento potencialmente poderia ser atualizado. Quando um host i atualizar um documento d, ele incrementará o número de versão $V_{d,i}[i]$ em um.

Sempre que os dois hosts i e j se conectam, eles trocam documentos atualizados, de modo que ambos obtêm novas versões dos documentos. Porém, antes de trocar documentos, os hosts precisam descobrir se as cópias são consistentes:

1. Se os vetores de versão forem os mesmos nos dois hosts – ou seja, para cada k, $V_{d,i}[k] = V_{d,j}[k]$ – então as cópias do documento d são idênticas.
2. Se, para cada k, $V_{d,i}[k] \leq V_{d,j}[k]$ e os vetores de versão não forem idênticos, então a cópia do documento d no host i é mais antiga que aquela no host j. Ou seja, a cópia do documento d no host j foi obtida por uma ou mais modificações da cópia do documento no host i. O host i substitui sua cópia de d, além de sua cópia do vetor de versão para d, com as cópias do host j.
3. Se houver um par de hosts k e m de modo que $V_{d,i}[k] < V_{d,j}[k]$ e $V_{d,i}[m] > V_{d,j}[m]$, então as cópias são *inconsistentes*; ou seja, a cópia de d em i contém atualizações realizadas pelo host k que ainda não foram propagadas ao host j e, de modo semelhante, a cópia de d em j contém atualizações realizadas pelo host m que não foram propagadas ao host i. Então, as cópias de d são inconsistentes, pois duas ou mais atualizações foram realizadas sobre d independentemente. A intervenção manual pode ser exigida para mesclar as atualizações.

O esquema vetor de versão foi projetado inicialmente para lidar com falhas nos sistemas de arquivos distribuídos. O esquema ganhou importância porque os computadores móveis normalmente armazenam cópias de arquivos que também estão presentes nos sistemas de servidor, com efeito, construindo um sistema de arquivos distribuído, que normalmente está desconectado. Outra aplicação do esquema é nos sistemas de groupware, em que os hosts são conectados periodicamente, em vez de continuamente, e precisam trocar documentos atualizados. O esquema vetor de versão também tem aplicações em bancos de dados replicados.

Porém, o esquema vetor de versão não consegue resolver o problema mais difícil e mais importante que surge de atualizações aos dados compartilhados – a reconciliação de cópias de dados inconsistentes. Muitas aplicações podem realizar a reconciliação automaticamente, executando em cada computador aquelas operações que realizaram atualizações em computadores remotos durante o período de desconexão. Essa solução funciona se as operações de atualização forem comutativas – ou seja, elas geram o mesmo resultado, independente da ordem em que são executadas. Técnicas alternativas podem estar disponíveis em certas aplicações; no pior dos casos, porém, deve ficar com os usuários a solução das inconsistências. Continua sendo uma área de pesquisa o tratamento automático dessa inconsistência e o auxílio de usuários na solução de inconsistências que não podem ser tratadas automaticamente.

Outro ponto fraco é que o esquema vetor de versão exige uma comunicação substancial entre um host móvel reconectando e a estação de suporte móvel desse host. As verificações de consistência podem ser adiadas até que os dados sejam necessários, embora esse atraso possa aumentar a inconsistência geral do banco de dados.

O potencial para desconexão e o custo da comunicação sem fio limita a praticidade das técnicas de processamento de transação discutidas no Capítulo 22 para sistemas distribuídos. Normalmente, é preferível deixar que os usuários preparem transações em hosts móveis, mas exigir que, em vez de executar as transações no local, eles submetam transações a um servidor para execução. As transações que se espalham por mais de um computador e que incluem um host móvel encaram o bloqueio a longo prazo durante o commit da transação, a menos que a desconectividade seja rara ou previsível.

Resumo

- O tempo desempenha um papel importante nos sistemas de banco de dados. Os bancos de dados são modelos do mundo real. Enquanto a maioria dos bancos de dados modela o estado do mundo real em um ponto no tempo (no momento atual), os bancos de dados temporais modelam os estados do mundo real no decorrer do tempo.
- Os fatos nas relações temporais possuem tempos associados quando são válidos, que podem ser representados como uma união de intervalos. As linguagens de consulta temporais simplificam a modelagem do tempo, além de consultas relacionadas ao tempo.
- Os bancos de dados espaciais estão encontrando uso cada vez maior hoje para armazenar os dados de projeto auxiliado por computador e também dados geográficos.

- Dados de projeto são armazenados principalmente como dados de vetor; os dados geográficos consistem em uma combinação de dados de vetor e rastreio. As restrições de integridade espacial são importantes para os dados de projeto.
- Os dados de vetor podem ser codificados como dados na primeira forma normal, ou podem ser armazenados usando estruturas não da primeira forma normal, como listas. As estruturas de índice de uso especial são particularmente importantes para acessar dados espaciais e processar consultas espaciais.
- Árvores-R são uma extensão multidimensional das árvores B; com variantes como árvores R* e árvores R+, elas provaram ser populares nos bancos de dados espaciais. As estruturas de índice que dividem o espaço em um padrão regular, como árvores quadráticas, ajudam no processamento de consultas de junção espacial.
- Os bancos de dados de multimídia estão crescendo em importância. Questões como recuperação baseada em semelhança e remessa de dados em velocidades garantidas atualmente são tópicos de pesquisa.
- Os sistemas de computação móveis se tornaram comuns, levando ao interesse nos sistemas de banco de dados que podem executar em tais sistemas. O processamento da consulta em tais sistemas pode envolver pesquisas em bancos de dados de servidor. O modelo de custo da consulta precisa incluir o custo da comunicação, incluindo o custo monetário e o custo da alimentação por bateria, que é relativamente alto para sistemas móveis.
- O broadcast é muito mais barato por destinatário que a comunicação ponto a ponto, e o broadcast de dados como os dados do mercado de ações ajuda os sistemas móveis a obterem dados a custo reduzido.
- A operação desconectada, o uso de dados de broadcast e o caching de dados são três questões importantes consideradas na computação móvel.

Termos de revisão

- Dados temporais
- Tempo valido
- Tempo de transação
- Relação temporal
- Relação bitemporal
- Universal Coordinated Time (UTC)
- Relação de snapshot
- Linguagens de consulta temporais
- Seleção temporal
- Projeção temporal
- Junção temporal
- Dados espaciais e geográficos
- Dados de Computer Aided Design (CAD)
- Dados geográficos
- Sistemas de informações geográficas
- Triangulação
- Bancos de dados de projeto
- Dados geográficos
- Dados de rastreio
- Dados de vetor
- Global Positioning System (GPS)
- Consultas espaciais
- Consultas de proximidade
- Consultas de vizinho mais próximo
- Consultas de região
- Junção espacial
- Indexação de dados espaciais
- Árvores k-d
- Árvores k-d-B
- Árvores quadráticas
  - Árvore quadrática PR
  - Árvore quadrática de região
- Árvores R
  - Caixa de contorno
  - Divisão quadrática
- Bancos de dados de multimídia
- Dados isócronos
- Dados de mídia contínua
- Recuperação baseada em semelhança
- Formatos de dados de multimídia
- Servidores de vídeo
- Computação móvel
  - Hosts móveis
  - Estações de suporte móveis
  - Célula
  - Repasse
- Consultas dependentes de local
- Dados de broadcast
- Consistência
  - Relatórios de invalidação
  - Esquema vetor de versão

Exercícios práticos

24.1 Quais são os dois tipos de tempo e qual é a diferença entre eles? Por que faz sentido ter os dois tipos de tempo associados a uma tupla?

24.2 Suponha que você tenha uma relação contendo as coordenadas x, y e nomes de restaurantes. Suponha também que as únicas consultas que serão solicitadas são da seguinte forma: a consulta especifica um ponto e pergunta se existe um restaurante exatamente nesse ponto. Que tipo de índice seria preferível, árvore R ou árvore B? Por quê?

24.3 Suponha que você tenha um banco de dados espacial que ofereça suporte para consultas de região (com regiões circulares), mas não consultas do vizinho mais próximo. Descreva um algoritmo que descobre o vizinho mais próximo utilizando várias consultas de região.

24.4 Suponha que você queira armazenar segmentos de linha em uma árvore R. Se um segmento de linha não estiver paralelo aos eixos, a caixa de contorno para ele pode ser grande, contendo uma área vazia grande.

- Descreva o efeito sobre o desempenho de ter grandes caixas de contorno em consultas que pedem segmentos de linha cruzando determinada região.
- Descreva rapidamente uma técnica para melhorar o desempenho de tais consultas e dê um exemplo de seu benefício. Dica: você pode dividir os segmentos em partes menores.

24.5 Dê um procedimento recursivo para calcular de modo eficiente a junção espacial de duas relações com índices de árvore R. (Dica: use caixas de contorno para verificar se as entradas de folha sob um par de nós internos podem se cruzar.)

24.6 Descreva como as idéias por trás da organização RAID (seção "RAID" do Capítulo 11) podem ser usadas em um ambiente de dados de banco de dados, em que ocasionalmente pode haver ruído que impeça a recepção de parte dos dados sendo transmitidos.

24.7 Defina um modelo para enviar dados repetidamente por broadcast, em que o meio de broadcast é modelado como um disco virtual. Descreva como o tempo de acesso e a taxa de transferência de dados para esse disco virtual diferem dos valores correspondentes para um disco rígido típico.

24.8 Considere um banco de dados de documentos em que todos os documentos são mantidos em um banco de dados central. Cópias de alguns documentos são mantidas em computadores móveis. Suponha que o computador móvel A atualize uma cópia do documento 1 enquanto está desconectado e, ao mesmo tempo, o computador móvel B atualize uma cópia do documento 2 enquanto está desconectado. Mostre como o esquema vetor de versão pode garantir a atualização correta do banco de dados central e computadores móveis quando um computador móvel é reconectado.

Exercícios

24.9 As dependências funcionais serão preservadas se uma relação for convertida para uma relação temporal pelo acréscimo de um atributo de tempo? Como o problema será tratado em um banco de dados temporal?

24.10 Considere os dados de vetor bidimensional em que os itens de dados não se sobrepõem. É possível converter esses dados de vetor para dados de rastreio? Se for, quais são as desvantagens de armazenar dados de rastreio obtidos por tal conversão, em vez dos dados de vetor originais?

24.11 Estude o suporte para dados espaciais oferecido pelo sistema de banco de dados que você utiliza e implemente o seguinte:

a. Um esquema para representar o local geográfico dos restaurantes junto com os recursos como cardápios no restaurante e a faixa de preços.
b. Uma consulta para encontrar restaurantes de preço moderado que servem comida indiana e estão dentro de um raio de 8 quilômetros da sua casa (considere qualquer local para a sua casa).
c. Uma consulta para encontrar, para cada restaurante, a distância do restaurante mais próximo que serve a mesma variedade de comida e com o mesma faixa de preços.

24.12 Que problemas podem ocorrer em um sistema de mídia contínua se os dados forem entregues muito lentamente ou muito rapidamente?

24.13 Liste três recursos principais da computação móvel por redes sem fio que sejam distintos dos sistemas distribuídos tradicionais.

24.14 Liste três fatores que precisam ser considerados na otimização da consulta para a computação móvel que não são considerados nos otimizadores de consulta tradicionais.

24.15 Dê um exemplo para mostrar que o esquema vetor de distância não garante a seriação. (Dica: use o exemplo do Exercício prático 24.8, supondo que os documentos 1 e 2 estão disponíveis nos computadores móveis A e B, e leve em conta a possibilidade de que um documento possa ser lido sem ser atualizado.)

Notas bibliográficas

Stam e Snodgrass [1988] e Soo [1991] oferecem estudos sobre o gerenciamento de dados temporais. Jensen *et al.* [1994] apresentam um glossário de conceitos de banco de dados temporal, visando unificar a terminologia. Tansel *et al.* [1993] é uma coleção de artigos sobre diferentes aspectos dos bancos de dados temporais. Chomicki [1995] apresenta técnicas para gerenciar restrições de integridade temporais.

Ian Heywood [2002] oferece um livro-texto sobre sistemas de informação geográficos. Samet [1995b] oferece uma visão geral da grande quantidade de trabalho sobre estruturas de índice espaciais. Uma descrição antiga da árvore quadrática é fornecida por Finkel e Bentley [1974]. Samet [1990] e Samet [1995b] descrevem diversas variantes das árvores quadráticas. Bentley [1975] descreve a árvore k-d, e Robinson [1981] descreve a árvore k-d-B. A árvore R foi apresentada originalmente em Guttman [1984]. Extensões da árvore R são apresentadas por Sellis *et al.* [1987], que descrevem a árvore R^+, e Beckmann *et al.* [1990], que descrevem a árvore R^*.

Brinkhoff *et al.* [1993] discutem uma implementação das junções espaciais usando árvores R. Lo e Ravishankar [1996] e Patel e DeWitt [1996] apresentam métodos baseados em particionamento para o cálculo de junções espaciais. Samet e Aref [1995] oferecem uma visão geral dos modelos de dados espaciais, operações espaciais e a integração de dados espaciais e não espaciais.

Revesz [2002] oferece um livro-texto na área de bancos de dados de restrição; os intervalos temporais e as regiões espaciais podem ser imaginados como casos especiais de restrições.

Samet [1995a] descreve questões de pesquisa em bancos de dados de multimídia. A indexação de dados de multimídia é discutida em Faloutsos e Lin [1995].

Dashti *et al.* [2003] oferecem um livro-texto sobre o projeto de servidor de mídia streaming, incluindo uma cobertura extensa da organização de dados em subsistemas de disco. Os servidores de vídeo são discutidos em Anderson *et al.* [1992], Rangan *et al.* [1992], Ozden *et al.* [1994], Freedman e DeWitt [1995] e Ozden *et al.* [1996b]. A tolerância a falhas é discutida em Berson *et al.* [1995] e Ozden *et al.* [1996a].

O gerenciamento de informações nos sistemas que incluem computadores móveis é estudado em Alonso e Korth [1993] e Imielinski e Badrinath [1994]. Imielinski e Korth [1996] apresentam uma introdução à computação móvel e uma coleção de artigos de pesquisa sobre o assunto.

O esquema vetor de versão para detectar inconsistência nos sistemas de arquivo distribuídos é descrito por Popek *et al.* [1981] e Parker *et al.* [1983].

CAPÍTULO 25

Processamento avançado de transações

Nos Capítulos 15, 16 e 17, apresentamos o conceito de uma transação, que é uma unidade de programa que acessa – e possivelmente atualiza – diversos itens de dados, e cuja execução garante a preservação das propriedades ACID. Discutimos nesses capítulos uma série de esquemas para garantir as propriedades ACID em um ambiente em que a falha pode ocorrer e as transações podem ser executadas simultaneamente.

Neste capítulo, vamos além dos esquemas básicos discutidos anteriormente, e abordamos conceitos avançados de processamento de transação, incluindo monitores de processamento de transação, fluxos de trabalho transacionais e processamento de transação no contexto do comércio eletrônico. Também abordamos bancos de dados na memória principal, bancos de dados de tempo real, transações de longa duração, transações aninhadas e transações de múltiplos bancos de dados.

Monitores de processamento de transação

Os **monitores de processamento de transação** (**monitores de PT**) são sistemas que foram desenvolvidos nas décadas de 1970 e 1980, inicialmente em resposta a uma necessidade de dar suporte a uma grande quantidade de terminais remotos (como terminais de reserva de viagens aéreas) por um único computador. Em sua forma original, o termo "*TP monitor*" inicialmente significava *monitor de teleprocessamento*.

Desde então, os monitores de PT evoluíram para oferecer o suporte básico para processamento distribuído de transações, e o termo monitor de PT adquiriu seu significado atual. O monitor de PT CICS da IBM foi um dos primeiros monitores de PT, e foi muito utilizado. Os monitores de PT da geração atual incluem Tuxedo e Top End (ambos agora da BEA Systems), Encina (da Transarc, que agora faz parte da IBM) e Transaction Server (da Microsoft).

As arquiteturas de servidor de aplicações Web, incluindo servlets, que estudamos anteriormente na seção "Servlets e JSP" do Capítulo 8, admitem muitos dos recursos dos monitores de PT e às vezes são chamados de "PT lite". Os servidores de aplicação Web são muito utilizados e suplantaram os monitores de PT para muitas aplicações. Porém, os conceitos por trás deles, que estudamos nesta seção, são basicamente os mesmos.

Arquiteturas de monitor de PT

Os sistemas de processamento de transação em grande escala são montados em torno de uma arquitetura cliente-servidor. Um modo de montar esses sistemas é ter um processo servidor para cada cliente; o servidor realiza autenticação e depois executa ações solicitadas pelo cliente. Esse **modelo de processo por cliente** é ilustrado na Figura 25.1a. O modelo apresenta vários problemas com relação à utilização da memória e velocidade de processamento:

- Os requisitos de memória por processo são altos. Mesmo que a memória para o código do programa seja compartilhada por todos os processos, cada processo consome memória para dados locais e descritores de arquivo abertos, além de sobrecarga do sistema operacional, como tabelas de página para suporte à memória virtual.
- O sistema operacional divide o tempo de CPU disponível entre os processos alternando entre eles; essa técnica é chamada de **multitarefa**. Cada **troca de contexto** entre

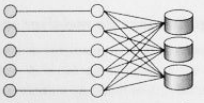
um processo e o seguinte possui uma sobrecarga de CPU considerável; até mesmo nos sistemas velozes de hoje, uma troca de contexto pode levar centenas de microssegundos.

Esses problemas podem ser evitados quando se tem um único processo de servidor para o qual todos os clientes remotos se conectam; esse modelo é chamado **modelo de servidor único**, ilustrado na Figura 25.1b. Os clientes remotos enviam solicitações ao processo servidor, que então as executa. Esse modelo também é usado nos ambientes cliente-servidor, em que os clientes enviam solicitações para um único processo de servidor. O processo servidor trata de tarefas, como autenticação do usuário, que normalmente seriam tratadas pelo sistema operacional. Para evitar o bloqueio de outros clientes quando processam uma solicitação longa para um cliente, o processo servidor é **multithreaded**: o processo servidor possui uma thread de controle para cada cliente e, com efeito, implementa sua própria multitarefa com baixa sobrecarga. Ele executa o código em favor de um cliente por um tempo, depois salva o contexto interno e passa para o código de outro cliente. Ao contrário da sobrecarga para a multitarefa completa, o custo de alternar entre as threads é baixo (normalmente, apenas alguns microssegundos).

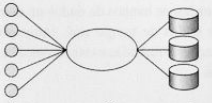
Os sistemas baseados no modelo de servidor único, como a versão original do monitor de PT CICS da IBM e os servidores de arquivos como Netware, da Novel, ofereceram com sucesso altas taxas de transação com recursos limitados. Entretanto, eles tinham problemas, especialmente quando várias aplicações acessavam o mesmo banco de dados:

- Como todas as aplicações são executadas como um único processo, não há proteção entre elas. Um bug em uma aplicação pode afetar também todas as outras aplicações. Seria melhor executar cada aplicação como um processo separado.
- Esses sistemas não são adequados para bancos de dados paralelos ou distribuídos, pois um processo servidor não pode ser executado em vários computadores ao mesmo tempo. (Porém, as threads concorrentes dentro de um processo podem ser admitidas em um sistema multiprocessador de memória compartilhada.) Essa é uma desvantagem séria nas grandes organizações, onde o processamento paralelo é crítico para o tratamento de grandes cargas de trabalho e os dados distribuídos estão se tornando cada vez mais comuns.

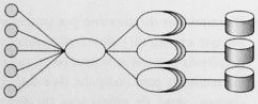
Uma maneira de resolver esses problemas é executar vários processos do servidor de aplicações que acessam um banco de dados comum e deixar que os clientes se comuniquem com a aplicação por meio de um único processo de comunicação, que encaminha as solicitações. Esse é o chamado **modelo muitos servidores, único roteador**, ilustrado na Figura 25.1c. Esse modelo admite processos de servidor independentes para múltiplas aplicações; além do mais, cada aplicação pode ter um pool de processos servi-



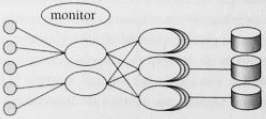
(a) Modelo de processo por cliente



(b) Modelo de servidor único



(c) Modelo muitos servidores, único roteador



(d) Modelo muitos servidores, muitos roteadores

Figura 25.1 Arquiteturas de monitor de PT.

dores, qualquer um deles podendo tratar da sessão de um cliente. A solicitação pode, por exemplo, ser direcionada para o servidor menos sobrecarregado em um pool. Como antes, cada processo servidor pode ser multithreaded, de modo que pode lidar com vários clientes simultaneamente. Como outra generalização, os servidores de aplicação podem ser executados em diferentes sites de um banco de dados paralelo ou distribuído, e o processo de comunicação pode lidar com a coordenação entre os processos.

Essa arquitetura também é muito usada nos servidores Web. Um servidor Web possui um processo principal, que recebe solicitações HTTP e depois tem a tarefa de entregar cada solicitação a um processo separado (escolhido dentre um pool de processos). Cada um dos processos, por si só, é multithreaded, de modo que pode lidar com múltiplas solicitações. O uso de linguagens de programação seguras, como Java, C# ou Visual Basic, permite que os servidores de aplicações Web protejam as threads contra erros em outras threads. Ao contrário, com uma linguagem tipo C ou C++, erros como os de alocação de memória em uma thread podem causar a falha de outras threads.

Uma arquitetura geral possui múltiplos processos, em vez de apenas um, para se comunicarem com os clientes. Os processos de comunicação com o cliente interagem com um ou mais processos roteadores, que direcionam suas solicitações para o servidor apropriado. Portanto, os monitores de PT de última geração possuem uma arquitetura diferente, denominada **modelo muitos servidores, muitos roteadores**, ilustrada na Figura 25.1d. Um processo controlador dá partida nos outros processos e supervisiona seu funcionamento. Tandem Pathway é um exemplo de um monitor de PT que usa essa arquitetura. Os processos do roteador normalmente são roteadores de rede que direcionam o tráfego voltado para o mesmo endereço Internet a diferentes computadores servidores, dependendo de onde vem o tráfego. O que aparece ao mundo exterior como sendo um único servidor, com um único endereço, pode ser uma coleção de servidores.

A estrutura detalhada de um monitor de PT é mostrada na Figura 25.2. Um monitor de PT faz mais do que simplesmente passar mensagens para os servidores de aplicações. Quando as mensagens chegam, elas podem ter de ser enfileiradas; assim, existe um **gerenciador de fila** para as mensagens que chegam. A fila pode ser uma **fila durável**, cujas entradas sobrevivem às falhas do sistema. O uso de uma fila durável ajuda a garantir que, uma vez recebidas e armazenadas na fila, as mensagens serão finalmente processadas, independente das falhas do sistema. O gerenciamento de autorização e de servidor de aplicações (por exemplo, partida do servidor e roteamento de mensagens aos servidores) são outras funções de um monitor de PT. Os monitores de PT normalmente oferecem facilidades de registro, recuperação e controle de concorrência, permitindo que os servidores de aplicações implementem as propriedades de transação ACID diretamente, se isso for exigido.

Finalmente, os monitores de PT também oferecem suporte para mensagens persistentes. Lembre-se de que as mensagens persistentes (seção "Modelos alternativos de processamento de transação" do Capítulo 22) oferecem uma garantia de que a mensagem será entregue se (e somente se) a transação for confirmada.

Além dessas facilidades, muitos monitores de PT também oferecem *facilidades de apresentação* para criar interfaces de menus/formulários para clientes burros, como terminais; essas facilidades não são mais importantes, pois os clientes burros quase não são mais utilizados.

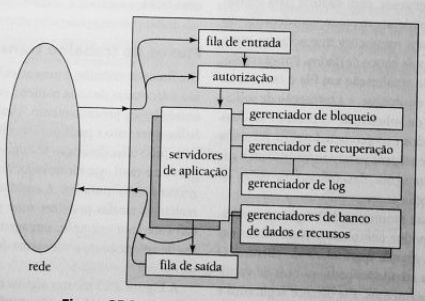


Figura 25.2 Componentes do monitor de PT.

Coordenação de aplicações usando monitores de PT

As aplicações hoje normalmente precisam interagir com vários bancos de dados. Elas também precisam interagir com sistemas legados, como sistemas de armazenamento de dados de uso especial, montados diretamente nos sistemas de arquivos. Finalmente, elas podem ter de se comunicar com usuários ou outras aplicações em sites remotos. Logo, elas também precisam interagir com os subsistemas de comunicação. É importante poder coordenar os acessos aos dados e implementar propriedades ACID para transações por tais sistemas.

Os monitores de PT modernos oferecem suporte para a construção e administração de aplicações grandes, montadas a partir de vários subsistemas, como bancos de dados, sistemas legados e sistemas de comunicação. Um monitor de PT trata de cada subsistema como um **gerenciador de recursos**, que oferece acesso transacional a algum conjunto de recursos. A interface entre o monitor de PT e o gerenciador de recursos é definida por um conjunto de primitivas de transação, como *begin_transaction*, *commit_transaction*, *abort_transaction* e *prepare_to_commit_transaction* (para o commit de duas fases). Naturalmente, o gerenciador de recursos também precisa oferecer outros serviços, como fornecer dados à aplicação.

A interface do gerenciador de recursos é definida pelo padrão X/Open Distributed Transaction Processing. Muitos sistemas de banco de dados têm suporte para os padrões X/Open e podem atuar como gerenciadores de recursos. Os monitores de PT – além de outros produtos, como sistemas SQL, que admitem os padrões X/Open – podem se conectar aos gerenciadores de recursos.

Além disso, os serviços fornecidos por um monitor de PT, como mensagens persistentes e filas duráveis, atuam como gerenciadores de recursos, com suporte para transações. O monitor de PT pode atuar como coordenador do commit de duas fases para transações que acessam esses serviços, além de sistemas de banco de dados. Por exemplo, quando uma transação de atualização em fila é executada, uma mensagem de saída é entregue, e a transação de solicitação é removida da fila de solicitação. O commit de duas fases entre o banco de dados e os gerenciadores de recursos para a fila durável e as mensagens persistentes ajudam a garantir que, independentemente das falhas, ou todas as ações ocorrem ou nenhuma ocorre.

Também podemos usar monitores de PT para administrar sistemas cliente-servidor complexos, consistindo em vários servidores e uma grande quantidade de clientes. O monitor de PT coordena atividades como pontos de verificação do sistema e desligamentos. Ele oferece segurança e autenticação de clientes. Além disso, administra pools de servidores acrescentando servidores ou removendo servidores sem interrupção do sistema de banco de dados. Finalmente, ele controla o escopo das falhas. Se um servidor falhar, o monitor de PT pode detectar essa falha, abortar as transações em andamento e reiniciar as transações. Se um nó falhar, o monitor de PT pode migrar transações para servidores em outros nós, novamente evitando transações incompletas. Quando os nós que falharam forem reiniciados, o monitor de PT pode governar a recuperação dos gerenciadores de recursos do nó.

Os monitores de PT podem ser usados para esconder as falhas do banco de dados nos sistemas replicados; os sistemas de backup remoto (seção "Sistemas de backup remoto" do Capítulo 17) são um exemplo de sistemas replicados. As solicitações de transação são enviadas ao monitor de PT, que repassa as mensagens para uma das réplicas do banco de dados (o site primário, no caso de sistemas de backup remotos). Se um site falhar, o monitor de PT pode rotear mensagens transparentemente para um site de backup, mascarando a falha do primeiro site.

Nos sistemas cliente-servidor, os clientes normalmente interagem com servidores por meio de um mecanismo de **chamada de procedimento remoto (RPC – Remote Procedure Call)**, em que um cliente invoca uma chamada de procedimento, que normalmente é executada no servidor, com os resultados enviados de volta ao cliente. Com relação ao código do cliente que invoca a RPC, a chamada se parece com uma invocação de chamada de procedimento. Os sistemas de monitor de PT, como Encina, oferecem uma interface **RPC transacional** para os seus serviços. Em uma interface desse tipo, o mecanismo de RPC oferece chamadas que podem ser usadas para delimitar uma série de chamadas de RPC dentro de uma transação. Assim, as atualizações realizadas por uma RPC são executadas dentro do escopo da transação, e podem ser revertidas se houver qualquer falha.

Fluxos de trabalho transacionais

Um fluxo de trabalho é uma atividade em que várias tarefas são executadas de uma maneira coordenada por diferentes entidades de processamento. Uma tarefa define algum trabalho a ser feito e pode ser especificada de várias maneiras, incluindo uma descrição textual em um arquivo ou mensagem de e-mail, um formulário, uma mensagem ou um programa de computador. A entidade de processamento que realiza as tarefas pode ser uma pessoa ou um sistema de software (por exemplo, um agente de correio, um programa de aplicação ou um sistema de gerenciamento de banco de dados).

A Figura 25.3 mostra alguns exemplos de fluxos de trabalho. Um exemplo simples é o de um sistema de e-mail. A

Aplicação de fluxo de trabalho	Tarefa típica	Entidade de processamento típica
roteamento de e-mail	mensagem de e-mail	servidores de correio
processamento de empréstimo	processamento de formulário	humanos, software de aplicação
processamento de ordem de compra	processamento de formulário	humanos, software de aplicação, SGBDs

Figura 25.3 Exemplos de fluxos de trabalho.

entrega de uma única mensagem de e-mail pode envolver vários sistemas de correio que recebem e encaminham a mensagem, até que ela alcance seu destino, onde é armazenada. Cada agente de correio realiza uma tarefa – encaminhando o correio para o próximo agente – e o trabalho de vários agentes de correio pode ser necessário a fim de direcionar o correio da origem até o seu destino. Outros termos usados na literatura de banco de dados e outras para se referir aos fluxos de trabalho são **fluxo de tarefa** e **aplicações multissistemas**. As tarefas do fluxo de trabalho às vezes são chamadas de **etapas**.

Em geral, os fluxos de trabalho podem envolver um ou mais humanos. Por exemplo, considere o processamento de um empréstimo. O fluxo de trabalho relevante aparece na Figura 25.4. A pessoa que deseja um empréstimo preenche um formulário, que é verificado pelo gerente responsável. Um funcionário que processa os pedidos de empréstimo verifica os dados no formulário, usando fontes como órgãos de referência de crédito. Quando todas as informações necessárias tiverem sido reunidas, o gerente pode decidir aprovar o empréstimo; essa decisão pode, então, ter de ser aprovada por um ou mais gerentes superiores, depois do que o empréstimo poderá ser feito. Cada pessoa aqui realiza uma tarefa; em um banco que não automatizou a tarefa de processamento de empréstimo, a coordenação das tarefas normalmente é executada pela passagem do pedido de empréstimo, incluindo anotações e outras informações, de um funcionário para o outro. Outros exemplos de fluxos de trabalho incluem o processamento de faturas de despesas, de ordens de compra e de transações de cartão de crédito.

Hoje, todas as informações relacionadas a um fluxo de trabalho provavelmente são armazenadas em um formato digital em um ou mais computadores, e, com o crescimento do uso de redes, as informações podem ser facilmente transferidas de um computador para outro. Logo, é viável para as organizações automatizarem seus fluxos de trabalho. Por exemplo, para automatizar as tarefas envolvidas no processamento de empréstimo, podemos armazenar a aplicação de empréstimo e as informações associadas em um banco de dados. O próprio fluxo de trabalho, então, envolve a passagem da responsabilidade de uma pessoa para a outra, e possivelmente ainda para programas que possam buscar automaticamente as informações exigidas. Os humanos podem coordenar suas atividades por meios do tipo e-mail.

Os fluxos de trabalho estão se tornando cada vez mais importantes por vários motivos dentro e também entre as organizações. Muitas organizações hoje possuem vários sistemas de software que precisam trabalhar juntos. Por exemplo, quando um funcionário ingressa em uma organização, as informações sobre o funcionário podem ter de ser fornecidas ao sistema de folha de pagamento, ao sistema de biblioteca, aos sistemas de autenticação que permitem que o usuário façam o logon nos computadores, a um sistema que gerencia contas de vale transporte ou vale alimentação, e assim por diante. As atualizações, por exemplo, quando o funcionário muda de cargo ou sai de um de-

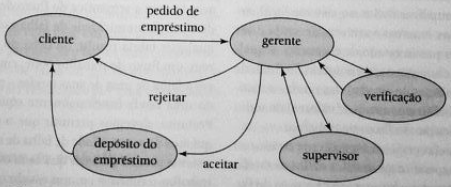


Figura 25.4 Fluxo de trabalho no processamento de empréstimo.

partamento, também precisam ser propagadas a todos os sistemas.

As organizações cada vez mais estão automatizando seus serviços; por exemplo, um fornecedor pode oferecer um sistema automatizado para os clientes fazerem pedidos. Várias tarefas podem ter de ser executadas quando um pedido é feito, desde reservar tempo em produção até criar o produto solicitado e os serviços para entrega do produto.

Temos de considerar duas atividades, em geral, para automatizar um fluxo de trabalho. A primeira é a **especificação de fluxo de trabalho**: detalhar as tarefas que precisam ser executadas e definir os requisitos de execução. O segundo problema é a **execução do fluxo de trabalho**, que precisamos fazer enquanto oferecemos as proteções dos sistemas de banco de dados tradicionais relacionadas à exatidão da computação e integridade/durabilidade dos dados. Por exemplo, não é aceitável que um pedido de empréstimo ou uma fatura se perca, ou que seja processado mais de uma vez, devido a uma falha do sistema. A idéia por trás dos fluxos de trabalho transacionais é usar e estender os conceitos das transações para o contexto dos fluxos de trabalho.

As duas atividades são complicadas pelo fato de que muitas organizações utilizam vários sistemas de processamento de informações gerenciados de forma independente que, na maioria dos casos, foram desenvolvidos separadamente para automatizar diferentes funções. As atividades de fluxo de trabalho podem exigir interações entre vários desses sistemas, cada um realizando uma tarefa, além de interações com seres humanos.

Diversos sistemas de fluxo de trabalho foram desenvolvidos nos últimos anos. Aqui, estudamos as propriedades dos sistemas de fluxo de trabalho em um nível relativamente abstrato, sem entrar nos detalhes de qualquer sistema em particular.

Especificação de fluxo de trabalho

Os aspectos internos de uma tarefa não precisam ser modelados para fins de especificação e gerenciamento de um fluxo de trabalho. Em uma visão abstrata de uma tarefa, esta pode utilizar os parâmetros armazenados em suas variáveis de entrada, apanhar e atualizar dados no sistema local, armazenar seus resultados em suas variáveis de saída e ser consultada com relação ao seu estado de execução. A qualquer momento, durante sua execução, o estado do fluxo de trabalho consiste na coleção de estados das tarefas constituintes do fluxo de trabalho e os estados (valores) de todas as variáveis na especificação do fluxo de trabalho.

A coordenação de tarefas pode ser especificada estática ou dinamicamente. Uma especificação estática define as tarefas – e as dependências entre elas – antes que a execução do fluxo de trabalho comece. Por exemplo, as tarefas em um fluxo de trabalho de fatura de despesas podem consistir em aprovações da fatura por uma secretária, um gerente e um contador, nessa ordem, e finalmente a remessa de um cheque. As dependências entre as tarefas podem ser simples – cada tarefa precisa ser completada antes que a próxima comece.

Uma generalização dessa estratégia é ter uma precondição para execução de cada tarefa do fluxo de trabalho, a fim de que todas as tarefas possíveis em um fluxo de trabalho e suas dependências sejam conhecidas com antecedência, mas que somente as tarefas cujas precondições forem atendidas sejam executadas. As precondições podem ser definidas por meio de dependências como as seguintes:

- **Estados de execução** de outras tarefas – por exemplo, "tarefa t_i não pode começar antes que a tarefa t_j tenha terminado", ou "tarefa t_i precisa abortar se a tarefa t_j tiver sido confirmada"
- **Valores de saída** de outras tarefas – por exemplo, "a tarefa t_i pode começar se a tarefa t_j retornar um valor maior do que 25", ou "a tarefa aprovação-do-gerente pode começar se a tarefa aprovação-da-secretária retornar um valor OK"
- **Variáveis externas** modificadas por eventos externos – por exemplo, "tarefa t_i não pode ser iniciada antes das 9 horas", ou "tarefa t_i precisa ser iniciada dentro de 24 horas a partir do término da tarefa t_j"

Podemos combinar as dependências por meio dos conectores lógicos comuns (or, and, not) para formar precondições de escalonamento complexas.

Um exemplo do escalonamento dinâmico das tarefas é um sistema de roteamento de e-mail. A tarefa seguinte a ser escalonada para determinada mensagem de correio depende do endereço de destino da mensagem e de quais roteadores intermediários estão funcionando.

Requisitos de atomicidade de falha de um fluxo de trabalho

O projetista do fluxo de trabalho pode especificar os requisitos de atomicidade de falha de um fluxo de trabalho de acordo com a semântica do fluxo de trabalho. A noção tradicional de atomicidade de falha exigiria que uma falha de qualquer tarefa resulte na falha do fluxo de trabalho. Porém, um fluxo de trabalho pode, em muitos casos, sobreviver à falha de uma de suas tarefas – por exemplo, executando uma tarefa funcionalmente equivalente em outro site. Portanto, devemos permitir que o projetista defina os requisitos de atomicidade de falha de um fluxo de trabalho. O sistema precisa garantir que cada execução de um fluxo de trabalho terminará em um estado que satisfaça os requisitos de atomicidade de falha definidos pelo projetista. Cha-

mamos esses estados de **estados de término aceitáveis** de um fluxo de trabalho. Todos os outros estados de execução de um fluxo de trabalho constituem um conjunto de **estados de término não aceitáveis**, em que os requisitos de atomicidade de falha podem ser violados.

Um estado de término aceitável pode ser designado como confirmado ou abortado. Um **estado de término aceitável confirmado** é um estado de execução em que os objetivos de um fluxo de trabalho foram alcançados. Ao contrário, um **estado de término aceitável abortado** é um estado de término válido em que um fluxo de trabalho deixou de alcançar seus objetivos. Se um estado de término aceitável abortado tiver sido alcançado, todos os efeitos indesejáveis da execução parcial do fluxo de trabalho precisam ser desfeitos de acordo com os requisitos de atomicidade de falha desse fluxo de trabalho.

Um fluxo de trabalho precisa alcançar um estado de término aceitável *mesmo na presença de falhas do sistema*. Assim, se um fluxo de trabalho estiver em um estado de término não aceitável no momento da falha, durante a recuperação do sistema ele precisa ser levado a um estado de término aceitável (seja ele abortado ou confirmado).

Por exemplo, no fluxo de trabalho de processamento de empréstimo, no estado final, ou o candidato de empréstimo é informado de que um empréstimo não pode ser feito ou o empréstimo é desembolsado. No caso de falhas como uma falha longa do sistema de verificação, o pedido de empréstimo poderia ser retornado ao solicitante do empréstimo com uma explicação apropriada; esse resultado constituiria um término aceitável abortado. Um término aceitável confirmado seria a aceitação ou a rejeição do pedido de empréstimo.

Em geral, uma tarefa pode confirmar e liberar seus recursos antes que o fluxo de trabalho alcance um estado de término. Porém, se a transação de multitarefa mais tarde for abortada, sua atomicidade de falha pode exigir que desfaçamos os efeitos das tarefas já completadas (por exemplo, subtransações confirmadas) executando tarefas de compensação (como subtransações). A semântica da compensação exige que uma transação de compensação por fim termine sua execução com sucesso, possivelmente após uma série de novas tentativas.

Em um fluxo de trabalho de processamento de fatura de despesas, por exemplo, um saldo de orçamento do departamento pode ser reduzido com base em uma aprovação inicial de uma fatura pelo gerente. Se a fatura mais tarde for rejeitada, seja por motivo de falha ou por outros motivos, o orçamento pode ter de ser restaurado por uma transação de compensação.

Execução de fluxos de trabalho

A execução das tarefas pode ser controlada por um coordenador humano ou pelo sistema de software chamado **sistema de gerenciamento de fluxo de trabalho**. Um sistema de gerenciamento de fluxo de trabalho consiste em um escalonador, agentes de tarefa e um mecanismo para consultar o estado do sistema de fluxo de trabalho. Um agente de tarefa controla a execução de uma tarefa por uma entidade de processamento. Um escalonador é um programa que processa fluxos de trabalho submetendo várias tarefas para execução, monitorando vários eventos e avaliando condições relacionadas às dependências entre tarefas. Um escalonador pode submeter uma tarefa para execução (a um agente de tarefa) ou pode solicitar que uma tarefa previamente submetida seja abortada. No caso de transações de múltiplos bancos de dados, as tarefas são subtransações, e as entidades de processamento são sistemas de gerenciamento de banco de dados locais. De acordo com as especificações de fluxo de trabalho, o escalonador impõe as dependências de escalonamento e é responsável por garantir que as tarefas alcancem estados de término aceitáveis.

Existem três técnicas arquitetônicas para o desenvolvimento de um sistema de gerenciamento de fluxo de trabalho. Uma **arquitetura centralizada** possui um único escalonador que escalona as tarefas para todos os fluxos de trabalho executando simultaneamente. Uma **arquitetura parcialmente distribuída** possui um escalonador instanciado para cada fluxo de trabalho. Quando as questões de execução simultânea podem ser separadas da função de escalonamento, essa segunda opção é a escolha natural. Uma **arquitetura totalmente distribuída** não possui escalonador, mas os agentes de tarefa coordenam sua execução comunicando-se entre si para satisfazer as dependências de tarefa e outros requisitos de execução de fluxo de trabalho.

Os sistemas de execução de fluxo de trabalho mais simples seguem a técnica totalmente distribuída que descrevemos e são baseados em mensagens. As mensagens podem ser implementadas por mecanismos de mensagem persistentes, para oferecer remessa garantida. Algumas implementações utilizam e-mail para as mensagens; essas implementações oferecem muitos dos recursos das mensagens persistentes, mas geralmente não garantem a atomicidade da remessa da mensagem e do commit da transação. Cada site possui um agente de tarefa que executa as tarefas recebidas por meio de mensagens. A execução também envolve a apresentação de mensagens às pessoas, que precisam então executar alguma ação. Quando uma tarefa for completada em um site e precisar ser processada em outro site, o agente de tarefa despacha uma mensagem ao próximo site. A mensagem contém todas as informações relevantes sobre a tarefa a ser realizada. Alguns sistemas de fluxo de trabalho baseados em mensagem são particularmente úteis em redes que podem estar desconectadas durante parte do tempo, como as redes dial-up.

A técnica centralizada é usada nos sistemas de fluxo de trabalho em que os dados são armazenados em um banco de dados central. O escalonador notifica vários agentes, como pessoas ou programas de computador, de que uma tarefa foi executada, e acompanha o término da tarefa. É mais fácil acompanhar o estado de um fluxo de trabalho com uma técnica centralizada do que com uma técnica totalmente distribuída.

O escalonador precisa garantir que um fluxo de trabalho terminará em um dos estados de término aceitável especificados. O ideal é que, antes de tentar executar um fluxo de trabalho, o escalonador examine esse fluxo de trabalho para verificar se o fluxo de trabalho pode terminar em um estado não aceitável. Se o escalonador não puder garantir que um fluxo de trabalho terminará em um estado aceitável, ele deverá rejeitar essas especificações sem tentar executar o fluxo de trabalho. Como exemplo, vamos considerar um fluxo de trabalho consistindo em duas tarefas representadas por subtransações S_1 e S_2, com os requisitos de atomicidade de falha indicando que ambas ou nenhuma das subtransações devem ser confirmadas. Se S_1 e S_2 não oferecerem estados preparada-para-confirmar (para um commit de duas fases) e além disso não tiverem transações de compensação, então é possível alcançar um estado em que uma subtransação é confirmada e a outra é abortada, e não existe um meio de trazer os dois para o mesmo estado. Portanto, essa especificação de fluxo de trabalho é **insegura** e deve ser rejeitada.

Verificações de segurança como a que descrevemos podem ser impossíveis ou impraticáveis de serem implementadas no escalonador; então, torna-se responsabilidade da pessoa que projetou a especificação do fluxo de trabalho garantir que os fluxos de trabalho sejam seguros.

Recuperação de um fluxo de trabalho

O objetivo da **recuperação do fluxo de trabalho** é impor uma atomicidade de falha dos fluxos de trabalho. Os procedimentos de recuperação precisam se certificar de que, se houver uma falha em qualquer um dos componentes de processamento de fluxo de trabalho (incluindo o escalonador), o fluxo de trabalho por fim alcançará um estado de término aceitável (seja abortado ou confirmado). Por exemplo, o escalonador poderia continuar processando após a falha e recuperação, como se nada tivesse acontecido, oferecendo assim a facilidade de recuperação direta. Caso contrário, o escalonador poderia abortar o fluxo de trabalho inteiro (ou seja, alcançar um dos estados de aborto globais). Nesse caso, algumas subtransações podem ter de ser confirmadas ou até mesmo submetidas para execução (por exemplo, subtransações de compensação).

Consideramos que as entidades de processamento envolvidas no fluxo de trabalho têm seus próprios sistemas de recuperação locais e tratam de suas falhas locais. Para recuperar o contexto do ambiente de execução, as rotinas de recuperação de falhas precisam restaurar a informação de estado do escalonador no momento da falha, incluindo as informações sobre os estados de execução de cada tarefa. Portanto, a informação de status apropriada precisa ser registrada em log, no armazenamento estável.

Também precisamos considerar o conteúdo das filas de mensagem. Quando um agente passa uma tarefa para outro, a passagem deve ser executada exatamente uma vez: se ela ocorrer duas vezes, uma tarefa pode ser executada duas vezes; se a passagem não ocorrer, a tarefa pode se perder. As mensagens persistentes (seção "Modelos alternativos de processamento de transação" do Capítulo 22) oferecem exatamente os recursos para garantir a passagem positiva e única.

Sistemas de gerenciamento de fluxo de trabalho

Os fluxos de trabalho normalmente são codificados à mão como parte dos sistemas de aplicação. Por exemplo, os sistemas de ERP (Enterprise Resource Planning), que ajudam a coordenar as atividades de uma empresa inteira, possuem diversos fluxos de trabalho embutidos.

O objetivo dos sistemas de gerenciamento de fluxo de trabalho é simplificar a construção de fluxos de trabalho e torná-los mais confiáveis, permitindo que sejam especificados em alto nível e executados de acordo com a especificação. Existem muitos sistemas comerciais de gerenciamento de fluxo de trabalho; alguns, como o FlowMark da IBM, são sistemas de gerenciamento de fluxo de trabalho de uso geral, enquanto outros são específicos a determinados fluxos de trabalho, como sistemas de processamento de pedidos ou relatório de bugs/falhas.

No mundo atual de organizações interconectadas, não é suficiente gerenciar fluxos de trabalho apenas dentro de uma organização. Os fluxos de trabalho que atravessam fronteiras organizacionais estão se tornando cada vez mais comuns. Por exemplo, considere um pedido feito por uma organização e comunicado a outra organização que atende o pedido. Em cada organização pode haver um fluxo de trabalho associado ao pedido, e é importante que os fluxos de trabalho possam interoperar, a fim de reduzir a intervenção humana.

A Workflow Management Coalition desenvolveu padrões para interoperação entre sistemas de fluxo de trabalho. Os esforços de padronização atuais utilizam XML como linguagem básica para a comunicação de informações sobre o fluxo de trabalho. Veja mais informações nas notas bibliográficas.

E-Commerce

E-commerce (comércio eletrônico) refere-se ao processo de executar várias atividades relacionadas ao comércio, por

meios eletrônicos, principalmente a Internet. Alguns tipos de atividades incluem:

- Atividades de pré-vendas, necessárias para informar ao comprador em potencial sobre o produto ou serviço sendo vendido.
- O processo de venda, que inclui negociações sobre preço e qualidade do serviço, além de outras questões contratuais.
- O mercado: quando existem vários vendedores e compradores para um produto, um mercado, como o de ações, ajuda a negociar o preço a ser pago pelo produto. Os leilões são usados quando existe um único vendedor e vários compradores, e os leilões reversos são usados quando existe um único comprador e vários vendedores.
- Pagamento pela venda.
- Atividades relacionadas à remessa do produto ou serviço. Alguns produtos e serviços podem ser remetidos pela Internet; para outros, a Internet é usada apenas para oferecer informações de remessa e rastrear remessas de produtos.
- Suporte ao cliente e serviço de pós-vendas.

Os bancos de dados são muito usados para dar suporte a essas atividades. Para algumas das atividades, o uso de bancos de dados é simples, mas existem questões interessantes de desenvolvimento de aplicações para as outras atividades.

E-catalogs

Qualquer site de e-commerce oferece aos usuários um catálogo dos produtos e serviços que o site fornece. Os serviços oferecidos por um e-catalog podem variar bastante.

No mínimo, um e-catalog precisa oferecer facilidades de navegação e busca para ajudar os clientes a encontrar o produto que estão procurando. Para ajudar a navegação, os produtos devem ser organizados em uma hierarquia intuitiva, de modo que alguns cliques nos hiperlinks podem levar clientes aos produtos em que estão interessados. As palavras-chave oferecidas pelo cliente (por exemplo, "câmera digital" ou "computador") devem agilizar o processo de localização dos produtos solicitados. Os e-catalogs também devem oferecer um meio para os clientes compararem facilmente alternativas dentre produtos concorrentes.

Os e-catalogs podem ser personalizados para o cliente. Por exemplo, um revendedor pode ter um acordo com uma grande empresa para fornecer alguns produtos com desconto. Um funcionário da empresa, vendo o catálogo para comprar produtos para a empresa, deverá ver preços com o desconto negociado, em vez dos preços normais. Devido a restrições legais sobre vendas de alguns tipos de itens, os clientes menores de idade, ou de certos estados ou países, não poderão ver itens que não possam ser legalmente vendidos para eles. Os catálogos também podem ser personalizados para usuários individuais, com base no histórico das compras passadas. Por exemplo, os clientes freqüentes podem receber descontos especiais sobre alguns itens.

O suporte de tal personalização requer informações do cliente e também informações especiais de preço/desconto e informações de restrição de vendas a serem armazenadas em um banco de dados. Há também desafios no suporte a taxas de transação muito altas, que normalmente são resolvidas pelo caching de resultados de consulta ou páginas Web geradas.

Mercados

Quando existem vários vendedores ou vários compradores (ou ambos) para um produto, um mercado ajuda na negociação do preço a ser pago pelo produto. Existem vários tipos de mercados:

- Em um sistema de **leilão reverso**, um comprador declara requisitos, e os vendedores propõem o fornecimento do item. O fornecedor que cotar o menor preço ganha. Em um sistema de proposta fechada, as propostas não se tornam públicas, enquanto em um sistema de proposta aberta, elas se tornam públicas.
- Em um **leilão**, existem vários compradores e um único vendedor. Para simplificar, considere que existe apenas uma instância de cada item sendo vendido. Os compradores fazem lances para os itens sendo vendidos, e quem fez o lance mais alto para um item compra o item por esse preço.

  Quando existem várias cópias de um item, as coisas se tornam mais complicadas: suponha que existem quatro itens e um comprador pode querer três cópias por $10 cada, enquanto outro deseja duas cópias por $13 cada. Não é possível satisfazer ambas as propostas. Se os itens não tiverem valor se não forem vendidos (por exemplo, passagens aéreas, que precisam ser vendidas antes que o avião saia), o vendedor simplesmente apanha o conjunto de propostas que maximiza a receita. Caso contrário, a decisão é mais complicada.
- Em uma **permuta**, como em uma permuta de ações, existem vários vendedores e vários compradores. Os compradores podem especificar o preço máximo que eles desejam pagar, enquanto os vendedores especificam o preço mínimo que desejam. Normalmente, há um *criador de mercado* que casa propostas de compra e venda, decidindo sobre o preço de cada negócio (por exemplo, o preço da proposta de venda).

Existem outros tipos de mercados mais complexos.

Entre os problemas de banco de dados no tratamento de mercados estão:

- Proponentes precisam ser autenticados antes que tenham permissão para propor.
- As propostas (compra ou venda) precisam ser registradas com segurança em um banco de dados. Além disso, precisam ser comunicadas rapidamente a outras pessoas envolvidas no mercado (como todos os compradores ou todos os vendedores), que podem ser numerosas.
- Os atrasos no envio de propostas podem levar a perdas financeiras para alguns participantes.
- Os volumes de negócios podem ser extremamente grandes em momentos de volatilidade do mercado de ações, ou no final dos leilões. Assim, bancos de dados de desempenho muito alto com grandes graus de paralelismo são usados para tais sistemas.

Estabelecimento de pedido

Depois que os itens tiverem sido selecionados (talvez por meio de um catálogo eletrônico) e o preço determinado (talvez por um mercado eletrônico), o pedido precisa ser estabelecido. O estabelecimento envolve o pagamento pelos bens e a remessa dos bens.

Um modo simples (porém, não seguro) de pagamento eletrônico é enviar um número de cartão de crédito. Existem dois problemas principais. Primeiro, é possível que haja fraude de cartão de crédito. Quando o comprador paga pelos bens físicos, as empresas podem garantir que o endereço para entrega combina com o cartão de crédito de quem o possui, de modo que ninguém mais possa receber os bens; para os bens entregues eletronicamente, essa verificação não é possível. Segundo, o vendedor precisa ser confiável para cobrar apenas pelo item combinado e não passar o número do cartão para pessoas não autorizadas, que poderão fazer uso indevido.

Vários protocolos estão disponíveis para pagamentos seguros, que evitam esses dois problemas listados. Além disso, eles oferecem melhor privacidade, por meio da qual o vendedor pode não receber quaisquer detalhes desnecessários sobre o comprador, e a companhia de cartão de crédito não recebe informações desnecessárias sobre os itens comprados. Todas as informações transmitidas precisam ser criptografadas, de modo que qualquer um que esteja interceptando os dados na rede não possa descobrir o conteúdo. A criptografia por chave pública/privada é muito usada para essa tarefa.

Os protocolos também precisam impedir os **ataques de pessoas no meio**, em que alguém pode personificar o banco ou a companhia de cartão de crédito, ou até mesmo o vendedor, ou comprador, e roubar informações secretas. A personificação pode ser perpetrada passando-se uma chave falsa como a chave pública de alguém (do banco, da companhia de cartão de crédito ou do comerciante ou comprador). A personificação é impedida por um sistema de **certificados digitais**, pelos quais as chaves públicas são assinadas por uma agência de certificação, cuja chave pública é bem conhecida (ou que, por sua vez, tem sua chave pública certificada por outra agência de certificação, e assim por diante, até uma chave bem conhecida). Pela chave pública bem conhecida, o sistema pode autenticar as outras chaves verificando os certificados na sequência reversa. Os certificados digitais já foram descritos, na seção "Certificados digitais" do Capítulo 8.

O **Secure Electronic Transaction** (SET) é um desses protocolos de pagamento seguro e exige várias rodadas de comunicação entre o comprador, o vendedor e o banco, a fim de garantir a segurança da transação.

Há também sistemas que oferecem maior anonimato, semelhante ao que é oferecido pelo dinheiro físico. O sistema de pagamento **DigiCash** é desse tipo. Quando um pagamento é feito em tal sistema, não é possível identificar o comprador. Ao contrário, a identificação de compradores é muito fácil com cartões de crédito, e até mesmo no caso da SET, é possível identificar o comprador com a cooperação da companhia de cartão de crédito ou do banco.

Bancos de dados na memória principal

Para permitir uma alta taxa de processamento de transação (centenas ou milhares de transações por segundo), temos de usar um hardware de alto desempenho e explorar o paralelismo. Porém, somente essas técnicas são insuficientes para obter tempos de resposta muito baixos, pois a E/S de disco continua sendo um gargalo – cerca de 10 milissegundos são necessários para cada E/S, e esse número não diminuiu em uma taxa comparável ao aumento nas velocidades do processador. A E/S de disco normalmente é o gargalo para leituras e também para commits de transação. A longa latência de disco (cerca de 10 milissegundos na média) aumenta não apenas o tempo para acessar um item de dados, mas também limita o número de acessos por segundo.

Podemos tornar o sistema de banco de dados menos voltado para o disco aumentando o tamanho do buffer de banco de dados. Os avanços na tecnologia de memória principal permitem construir grandes memórias principais a um custo relativamente baixo. Hoje, sistemas comerciais de 64 bits podem admitir memórias principais de dezenas de gigabytes.

Para algumas aplicações, como controle de tempo real, é necessário armazenar dados na memória principal para atender requisitos de desempenho. O tamanho de memória exigido para a maioria desses sistemas não é excepcionalmente grande, embora existam pelo menos algumas aplica-

ções que exigem múltiplos gigabytes de dados para estarem residentes na memória. Como os tamanhos de memória vêm crescendo a um ritmo muito rápido, um número cada vez maior de aplicações poderá ter seus dados armazenados na memória principal.

Grandes memórias principais permitem o processamento mais rápido de transações, pois os dados estão residentes na memória. Porém, ainda existem limitações relacionadas ao disco:

- Registros de log precisam ser escritos no armazenamento estável antes que uma transação seja confirmada. O desempenho melhorado que é possível com uma memória principal grande pode resultar em um gargalo no processo de registro de log. Podemos reduzir o tempo de commit criando um buffer de log estável na memória principal, usando RAM não volátil (implementada, por exemplo, por memória alimentada por bateria). A sobrecarga imposta pelo registro de log também pode ser reduzida pela técnica de *commit em grupo*, discutida mais adiante nesta seção. O throughput (número de transações por segundo) ainda é limitado pela taxa de transferência de dados do disco de log.
- Os blocos de buffer marcados como modificados pelas transações confirmadas ainda precisam ser escritos para que a quantidade de log que precisa ser reproduzida no momento da recuperação seja diminuída. Se a taxa de atualização for extremamente alta, a taxa de transferência de dados pode se tornar um gargalo.
- Se o sistema falhar, toda a memória principal será perdida. Na recuperação, o sistema possui um buffer de banco de dados vazio, e os itens de dados precisam ser lidos do disco quando forem acessados. Portanto, mesmo depois que a recuperação estiver completa, leva-se algum tempo antes que o banco de dados seja totalmente carregado na memória principal e o processamento de alta velocidade das transações possa retomar.

Por outro lado, um banco de dados da memória principal oferece oportunidades para otimizações:

- Como a memória é mais dispendiosa do que o espaço em disco, estruturas de dados internas nos bancos de dados da memória principal precisam ser projetadas para reduzir os requisitos de espaço. Porém, as estruturas de dados podem ter ponteiros cruzando várias páginas, diferente daqueles nos bancos de dados de disco, em que o custo de E/S para atravessar várias páginas seria excessivamente alto. Por exemplo, estruturas de árvore nos bancos de dados da memória principal podem ser relativamente profundas, ao contrário das árvores B+, mas devem minimizar os requisitos de espaço.
- Não há necessidade de fixar as páginas de buffer na memória antes que os dados sejam acessados, pois as páginas de buffer nunca serão substituídas.
- As técnicas de processamento de consulta devem ser projetadas para reduzir a sobrecarga de espaço, de modo que os limites da memória principal não sejam excedidos enquanto uma consulta está sendo avaliada; essa situação resultaria em paginação para a área de swap e atrasaria o processamento da consulta.
- Quando o gargalo de E/S de disco for removido, operações como bloqueio e tranca podem se tornar gargalos. Esses gargalos precisam ser eliminados pelas melhorias na implementação dessas operações.
- Os algoritmos de recuperação podem ser otimizados, pois as páginas raramente precisam ser escritas para criar espaço para outras páginas.

TimesTen e DataBlitz são dois produtos de banco de dados da memória principal que exploram várias dessas otimizações, enquanto o banco de dados Oracle acrescentou recursos especiais para dar suporte a memórias principais muito grandes. Oferecemos informações adicionais sobre bancos de dados da memória principal nas referências das notas bibliográficas.

O processo de confirmação de uma transação T exige que esses registros sejam escritos no armazenamento estável:

- Todos os registros de log associados a T que não foram enviados para o armazenamento estável
- O registro de log <T commit>

Essas operações de saída constantemente exigem a saída de blocos que estão preenchidos apenas parcialmente. Para garantir que blocos quase cheios sejam enviados, usamos a técnicas de **commit em grupo**. Em vez de tentar confirmar T quanto T concluir, o sistema espera até que várias transações tenham terminado, ou até que um certo período de tempo tenha se passado desde que uma transação completou sua execução. Depois, ele confirma o grupo de transações que estão aguardando, juntas. Os blocos escritos no log do armazenamento estável teriam registros de várias transações. Com a escolha cuidadosa do tamanho de grupo e tempo de espera máximo, o sistema pode garantir que os blocos estejam cheios quando forem escritos no armazenamento estável sem fazer com que as transações esperem excessivamente. Essa técnica resulta, na média, em menos operações de saída por transação confirmada.

Embora o commit em grupo reduza a sobrecarga imposta pelo logging, ele resulta em um ligeiro atraso no commit de transações que realizam atualizações. A espera pode se tornar muito pequena (digamos, 10 milissegundos), o que é aceitável para muitas aplicações. Essas esperas podem ser

eliminadas se os discos ou os controladores de disco admitirem buffers de RAM não voláteis para operações de escrita. As transações podem ser confirmadas assim que a escrita seja realizada no buffer de RAM não volátil. Nesse caso, não há necessidade de commit em grupo.

Observe que o commit em grupo é útil até mesmo em bancos de dados com dados residentes em disco.

Sistemas de transação em tempo real

As restrições de integridade que consideramos até aqui dizem respeito aos valores armazenados no banco de dados. Em certas aplicações, as restrições incluem **prazos** pelos quais uma tarefa precisa ser concluída. Alguns exemplos dessas aplicações incluem gerenciamento de fábrica, controle de tráfego e escalonamento. Quando os prazos são incluídos, a exatidão de uma execução não é mais unicamente uma questão de consistência de banco de dados. Em vez disso, estamos preocupados com quantos prazos são perdidos, e por quanto tempo eles foram perdidos. Os prazos são caracterizados da seguinte forma:

- **Prazo rígido.** Problemas sérios, como falha do sistema, podem ocorrer se uma tarefa não for completada pelo seu prazo.
- **Prazo firme.** A tarefa tem valor zero se for completada após o prazo.
- **Prazo flexível.** A tarefa possui valor cada vez menor se for completada após o prazo, com o valor tendendo a zero à medida que o grau de atraso aumenta.

Os sistemas com prazos são chamados **sistemas de tempo real**.

O gerenciamento de transações em sistemas de tempo real precisam levar em conta os prazos. Se o protocolo de controle de concorrência determinar que uma transação T_i deve esperar, ele pode fazer com que T_i perca o prazo. Nesses casos, pode ser preferível apropriar-se da transação que mantém o bloqueio e permitir que T_i prossiga. Porém, a apropriação precisa ser usada com cuidado, pois o tempo perdido pela transação apropriada (devido à reversão e reinício) pode fazer com que a transação perca seu prazo. Infelizmente, é difícil determinar se o rollback ou a espera é preferível em determinada situação.

Uma dificuldade importante no suporte a restrições de tempo real surge da variância no tempo de execução da transação. No melhor caso, todos os acessos a dados referenciam dados no buffer de banco de dados. No pior dos casos, cada acesso faz com que uma página de buffer seja escrita em disco (precedida pelos registros de log necessários), seguida pela leitura em disco da página contendo os dados a serem acessados. Como os dois ou mais acessos ao disco, exigidos no pior caso, levam um tempo com várias ordens de grandeza maior que as referências à memória principal, exigidas no melhor caso, o tempo de execução da transação mal poderá ser estimado se os dados estiverem residentes em disco. Logo, os bancos de dados da memória principal normalmente são utilizados se restrições de tempo real tiverem de ser atendidas.

Entretanto, mesmo que os dados estejam residentes na memória principal, as variâncias no tempo de execução surgem de esperas de bloqueio, abortos de transação e assim por diante. Os pesquisadores dedicaram um esforço considerável para o controle de concorrência para bancos de dados de tempo real. Eles estenderam os protocolos de bloqueio para oferecer maior prioridade a transações com prazos curtos. Eles descobriram que os protocolos de concorrência otimistas funcionam bem nos bancos de dados de tempo real; ou seja, esses protocolos resultam em menos prazos perdidos do que até mesmo os protocolos de bloqueio estendidos. As notas bibliográficas oferecem referências à pesquisa na área de bancos de dados de tempo real.

Em sistemas de tempo real, os prazos, em vez da velocidade absoluta, são a questão mais importante. O projeto de um sistema de tempo real envolve garantir que haja poder de processamento suficiente para atender prazos sem exigir recursos de hardware excessivos. Alcançar esse objetivo, apesar da variância no tempo de execução resultante do gerenciamento de transações, continua sendo um problema desafiador.

Transações de longa duração

O conceito de transação foi desenvolvido inicialmente no contexto das aplicações de processamento de dados, em que a maioria das transações não é interativa e possui curta duração. Embora as técnicas apresentadas aqui e antes nos Capítulos 15, 16 e 17 funcionem bem nessas aplicações, problemas sérios surgem quando esse conceito é aplicado aos sistemas de banco de dados que envolvem interação humana. Essas transações possuem estas propriedades-chave:

- **Longa duração.** Quando uma pessoa interage com uma transação ativa, essa transação se torna uma **transação de longa duração** do ponto de vista do computador, pois o tempo de resposta humano é lento em relação à velocidade do computador. Além do mais, em aplicações de projeto, a atividade humana pode envolver horas, dias ou um período ainda mais longo. Assim, as transações podem ser de longa duração em termos humanos, assim como em termos de máquina.
- **Exposição de dados não confirmados.** Os dados gerados e exibidos a um usuário por uma transação de longa

duração são não confirmados, pois a transação pode abortar. Assim, os usuários – e, como resultado, outras transações – podem ser forçados a ler dados não confirmados. Se diversos usuários estiverem cooperando em um projeto, as transações do usuário podem ter de trocar dados antes da confirmação da transação.

- **Subtarefas.** Uma transação interativa pode consistir em um conjunto de subtarefas iniciadas pelo usuário. O usuário pode querer abortar uma subtarefa sem necessariamente causar o aborto da transação inteira.
- **Facilidade de recuperação.** É inaceitável abortar uma transação interativa de longa duração devido a uma falha do sistema. A transação ativa precisa ser recuperada a um estado que existia pouco tempo antes da falha, para que relativamente pouco trabalho humano seja perdido.
- **Desempenho.** O bom desempenho em um sistema de transação interativo é definido como tempo de resposta rápido. Essa definição está em contraste à de um sistema não interativo, em que o alto throughput (número de transações por segundo) é o objetivo. Os sistemas com alto throughput fazem uso eficiente dos recursos do sistema, porém, no caso de transações interativas, o recurso mais dispendioso é o usuário. Se a eficiência e a satisfação do usuário tiverem de ser otimizadas, o tempo de resposta precisa ser rápido (do ponto de vista humano). Nos casos em que uma tarefa leva muito tempo, o tempo de resposta deverá ser previsível (ou seja, a variância nos tempos de resposta deve ser baixa), para que os usuários possam administrar bem o seu tempo.

Nas Seções "Execuções não seriáveis" a "Questões de implementação", veremos por que essas cinco propriedades são incompatíveis com as técnicas apresentadas até aqui e discutiremos como essas técnicas podem ser modificadas para acomodar transações interativas de longa duração.

Execuções não seriáveis

As propriedades sobre as quais discutimos tornam impraticável impor o requisito usado nos capítulos anteriores, de que somente escalonamentos seriáveis sejam permitidos. Cada um dos protocolos de controle de concorrência do Capítulo 16 possui efeitos adversos sobre transações de longa duração:

- **Bloqueio de duas fases (2PL).** Quando um bloqueio não pode ser concedido, a transação que o solicita é forçada a esperar até que o item de dados em questão seja desbloqueado. A duração dessa espera é proporcional à duração da transação que mantém o bloqueio. Se o item de dados estiver bloqueado por uma transação de curta duração, esperamos que o tempo de espera seja curto (exceto no caso de impasse ou carga extraordinária no sistema). Porém, se o item de dados for bloqueado por uma transação de longa duração, a espera será de longa duração. Tempos de espera longos levam a um tempo de resposta maior e maiores chances de impasse.
- **Protocolos baseados em gráfico.** Os protocolos baseados em gráfico permitem que os bloqueios sejam liberados antes do que nos protocolos de bloqueio de duas fases, e impedem o impasse. Porém, eles impõem uma ordenação sobre os itens de dados. As transações precisam bloquear itens de dados de uma maneira coerente com a ordenação. Como resultado, uma transação pode ter de bloquear mais dados do que precisa. Além do mais, uma transação precisa manter um bloqueio até que não haja mais chance de que o bloqueio seja necessário novamente. Assim, as esperas de bloqueio de longa duração provavelmente ocorrerão.
- **Protocolos baseados em timestamp.** Os protocolos de timestamp nunca exigem que uma transação espere. Porém, eles exigem que as transações abortem sob certas circunstâncias. Se uma transação de longa duração for abortada, uma quantidade substancial de trabalho será perdida. Para transações não interativas, esse trabalho perdido é um problema de desempenho. Para transações interativas, o problema também é de satisfação do usuário. É altamente indesejável que um usuário descubra que o trabalho de várias horas foi desfeito.
- **Protocolos de validação.** Assim como os protocolos baseados em timestamp , os protocolos de validação impõem a seriação abortando a transação.

Assim, parece que a imposição da seriação resulta em esperas de longa duração, no aborto de transações de longa duração ou em ambos. Existem resultados teóricos, citados nas notas bibliográficas, que substanciam essa conclusão.

Outras dificuldades com a imposição da seriação surgem quando consideramos questões de recuperação. Antes, discutimos o problema de reversão (rollback) em cascata, em que o aborto de uma transação pode levar ao aborto de outras transações. Esse fenômeno é indesejável, principalmente para transações de longa duração. Se o bloqueio for usado, os bloqueios exclusivos terão de ser mantidos até o final da transação, isso se a reversão em cascata tiver de ser evitada. Contudo, essa manutenção de bloqueios exclusivos aumenta a extensão do tempo de espera da transação.

Assim, parece que a imposição da atomicidade da transação precisa levar a uma probabilidade maior de esperas de longa duração ou criar uma possibilidade de reversão em cascata.

Essas considerações são a base para os conceitos alternativos de exatidão das execuções simultâneas e recuperação de transação, que consideramos no restante desta seção.

Controle de concorrência

O objetivo fundamental do controle de concorrência de banco de dados é garantir que a execução simultânea de transações não resulte em uma perda de consistência de banco de dados. O conceito de seriação pode ser usado para alcançar esse objetivo, pois todos os escalonamentos seriáveis preservam a consistência do banco de dados. Porém, nem todos os escalonamentos que preservam a consistência do banco de dados são seriáveis. Como um exemplo, considere novamente um banco de dados bancário, consistindo em duas contas A e B, com o requisito de consistência de que a soma $A + B$ seja preservada. Embora o escalonamento da Figura 25.5 não seja seriável em conflito, apesar disso ele preserva a soma de $A + B$. Ele também ilustra dois pontos importantes sobre o conceito de exatidão sem seriação.

- A exatidão depende das restrições de consistência específicas para o banco de dados.
- A exatidão depende das propriedades das operações realizadas por cada transação.

Em geral, não é possível realizar uma análise automática das operações de baixo nível por transações e verificar seu efeito sobre as restrições de consistência do banco de dados. Porém, existem técnicas simples. Uma delas é usar as restrições de consistência do banco de dados como base para uma divisão do banco de dados em sub-bancos de dados, em que a concorrência possa ser controlada separadamente. Outra é tratar algumas operações além de **read** e **write** como operações de baixo nível fundamentais e estender o controle de concorrência para lidar com elas.

As notas bibliográficas referenciam outras técnicas para garantir a consistência sem exigir a seriação. Muitas dessas técnicas exploram variantes do controle de concorrência multiversão (ver seção "Recuperação com transações concorrentes" do Capítulo 17). Para aplicações de processamento de dados mais antigas, que só precisam de uma versão, os protocolos multiversão impõem um alto overhead de espaço para armazenar as versões extras. Como muitas das novas aplicações de banco de dados exigem a manutenção de versões de dados, as técnicas de controle de concorrência que exploram versões multiplas são práticas.

Transações aninhadas e multinível

Uma transação de longa duração pode ser vista como uma coleção de subtarefas ou subtransações relacionadas. Estruturando uma transação como um conjunto de subtransações, somos capazes de melhorar o paralelismo, pois pode ser possível executar várias subtransações em paralelo. Além do mais, é possível lidar com a falha de uma subtransação (devido a aborto, falha do sistema e assim por diante) sem ter de reverter a transação de longa duração inteira.

Uma transação aninhada ou multinível T consiste em um conjunto $T = \{t_1, t_2, \ldots, t_n\}$ de subtransações e uma ordem parcial P sobre T. Uma subtransação t_i em T pode abortar sem forçar T a abortar. Em vez disso, T pode reiniciar t_i ou simplesmente escolher não executar t_i. Se t_i confirmar, essa ação não torna t_i permanente (diferente da situação no Capítulo 17). Em vez disso, t_i *confirma para* T, e ainda pode abortar (ou exigir compensação – ver próxima seção) se T abortar. Uma execução de T não deve violar a ordem parcial P. Ou seja, se uma aresta $ti \rightarrow t_j$ aparecer no gráfico de precedência, então $t_j \rightarrow t_i$ não deverá estar no fechamento transitivo de P.

O aninhamento pode ter vários níveis de profundidade, representando uma subdivisão de uma transação em subtarefas, sub-subtarefas e assim por diante. No nível de aninhamento mais baixo, temos as operações de banco de dados padrão **read** e **write**, que usamos anteriormente.

Se uma subtransação de T tiver permissão para liberar bloqueios ao terminar, T é considerada uma **transação multinível**. Quando uma transação multinível representa

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Figura 25.5 Um escalonamento seriável sem conflito.

uma atividade de longa duração, a transação às vezes é conhecida como uma **saga**. Como alternativa, se os bloqueios mantidos por uma subtransação t_i de T forem automaticamente atribuídos a T no término de t_i, T será considerada uma **transação aninhada**.

Embora o principal valor prático das transações multinível surja em transações complexas, de longa duração, usaremos o exemplo simples da Figura 25.5 para mostrar como o aninhamento pode criar operações de nível superior, que podem melhorar a concorrência. Reescrevemos a transação T_1, usando subtransações $T_{1.1}$ e $T_{1.2}$, que realizam as operações de incremento ou decremento:

- T_1 consiste em
  - $T_{1.1}$, que subtrai 50 de A
  - $T_{1.2}$, que soma 50 a B

De modo semelhante, reescrevemos a transação T_2 usando as subtransações $T_{2.1}$ e $T_{2.2}$, que também realizam operações de incremento e decremento:

- T_2 consiste em
  - $T_{2.1}$, que subtrai 10 de B
  - $T_{2.2}$, que soma 10 a A

Nenhuma ordenação é especificada sobre $T_{1.1}$, $T_{1.2}$, $T_{2.1}$ e $T_{2.2}$. Qualquer execução dessas subtransações gerará um resultado correto. O escalonamento da Figura 25.5 corresponde ao escalonamento <$T_{1.1}$, $T_{2.1}$, $T_{1.2}$, $T_{2.2}$>.

Transações de compensação

Para reduzir a freqüência da espera de longa duração, arrumamos para que as atualizações não confirmadas sejam expostas a outras transações com execução simultânea. Na realidade, as transações multinível podem permitir essa exposição. Porém, a exposição de dados não confirmados cria o potencial para reversões em cascata. O conceito de **transações de compensação** nos ajuda a tratar desse problema.

Considere que a transação T seja dividida em várias subtransações $t_1, t_2, \ldots, t_n$. Após uma subtransação t_i ser confirmada, ela libera seus bloqueios. Agora, se a transação de nível externo T tiver de ser abortada, o efeito de suas subtransações precisa ser desfeito. Suponha que as subtransações $t_1, \ldots, t_k$ tenham sido confirmadas, e que t_{k+1} estava sendo executada quando foi tomada a decisão de abortar. Podemos desfazer os efeitos de t_{k+1} abortando essa subtransação. Porém, não é possível abortar as subtransações $t_1, \ldots, t_k$, pois elas já foram confirmadas.

Em vez disso, executamos uma nova subtransação ct_i, denominada *transação de compensação*, para desfazer o efeito de uma subtransação t_i. Cada subtransação t_i precisa ter uma transação de compensação ct_i. As transações de compensação precisam ser executadas na ordem inversa $ct_k, \ldots, ct_1$. Aqui estão vários exemplos de compensação:

- Considere o escalonamento da Figura 25.5, que mostramos como sendo correto, embora não seriável em conflito. Cada subtransação libera seus bloqueios quando termina sua execução. Suponha que T_2 falhe logo antes de terminar, após $T_{2.2}$ ter liberado seus bloqueios. Então, executamos uma transação de compensação para $T_{2.2}$, que subtrai 10 de A e uma transação de compensação para $T_{2.1}$, que soma 10 a B.
- Considere uma inserção em banco de dados pela transação T_i que, como efeito colateral, faça com que um índice de árvore B+ seja atualizado. A operação de inserção pode ter modificado vários nós do índice de árvore B+. Outras transações podem ter lido esses nós no acesso a dados diferentes daqueles do registro inserido por T_i. Como na seção "Técnicas de recuperação avançadas" do Capítulo 17, podemos desfazer a inserção excluindo o registro inserido por T_i. O resultado é uma árvore B+ correta, consistente, mas não é necessariamente aquela com a estrutura exata que tínhamos antes de T_i ser iniciada. Assim, a exclusão é uma ação de compensação para a inserção.
- Considere uma transação de longa duração T_1 representando uma reserva de viagem. A transação T possui três subtransações: $T_{1.1}$, que faz as reservas aéreas; $T_{1.2}$, que reserva carros alugados; e $T_{1.3}$, que reserva um quarto de hotel. Suponha que o hotel cancele a reserva. Em vez de desfazer toda T_1, compensamos a falha de $T_{1.3}$ excluindo a antiga reserva do hotel e fazendo uma nova.

Se o sistema falhar no meio da execução de uma transação de nível externo, suas subtransações precisam ser revertidas quando ela se recuperar. As técnicas descritas na seção "Técnicas de recuperação avançadas" do Capítulo 17 podem ser usadas para essa finalidade.

A compensação pela falha de uma transação requer que a semântica da transação que falhou seja usada. Para certas operações, como incremento ou inserção em uma árvore B+, a compensação correspondente é facilmente definida. Para transações mais complexas, os programadores de aplicação podem ter de definir a forma correta da compensação no momento em que a transação é codificada. Para transações interativas complexas, pode ser necessário que o sistema interaja com o usuário para determinar a forma apropriada de compensação.

Questões de implementação

Os conceitos de transação discutidos nesta seção criam sérias dificuldades para implementação. Apresentamos algu-

mas delas aqui, e discutimos como podemos resolver esses problemas.

As transações de longa duração precisam sobreviver a falhas do sistema. Podemos garantir que isso aconteça realizando um **redo** sobre subtransações confirmadas, e realizando ou um **undo** ou a compensação para quaisquer subtransações de curta duração que estivessem ativas no momento da falha. Porém, essas ações resolvem apenas parte do problema. Nos sistemas de banco de dados típicos, dados internos do sistema, como tabelas de bloqueio e timestamps de transação, são mantidos no armazenamento volátil. Para uma transação de longa duração ser retomada após uma falha, esses dados precisam ser restaurados. Portanto, é necessário registrar em log não apenas mudanças no banco de dados, mas também mudanças nos dados internos do sistema pertencentes a transações de longa duração.

O logging de atualizações se torna mais complexo quando existem certos tipos de itens de dados no banco de dados. Um item de dados pode ser um projeto CAD, texto de um documento ou outra forma de projeto composto. Esses itens de dados são fisicamente grandes. Assim, armazenar os valores antigo e novo do item de dados em um registro de log é indesejável.

Existem duas técnicas que reduzem a sobrecarga de garantir a facilidade de recuperação de grandes itens de dados:

- **Logging de operação.** Somente a operação realizada sobre o item de dados e o nome do item de dados são armazenados no log. O logging de operação também é chamado **logging lógico**. Para cada operação, é preciso haver uma operação inversa. Realizamos o **undo** usando a operação inversa e **redo** usando a própria operação. A recuperação por meio do logging de operação é mais difícil, pois **redo** e **undo** não são coerentes. Além disso, o uso de logging lógico para uma operação que atualiza várias páginas se torna bem mais complicado pelo fato de que algumas, mas não todas as páginas atualizadas podem ter sido escritas no disco, de modo que é difícil aplicar o **redo** ou o **undo** da operação sobre a imagem do disco durante a recuperação.

  O uso do logging de redo físico e o logging de undo lógico, conforme descrevemos na seção "Técnicas de recuperação avançadas" do Capítulo 17, oferece os benefícios de concorrência do logging lógico enquanto evita essas armadilhas.
- **Logging e paginação de sombra.** O logging é usado para modificações em pequenos itens de dados, mas grandes itens de dados normalmente se tornam recuperáveis por meio de uma técnica de cópia de sombra (ver seção "Implementação de atomicidade e durabilidade" do Capítulo 15). Quando usamos a sombra, é possível reduzir a sobrecarga mantendo cópias apenas daquelas páginas que estão realmente modificadas. Independente da técnica utilizada, as complexidades introduzidas pelas transações de longa duração e grandes itens de dados complicam o processo de recuperação. Assim, é importante permitir que certos dados não críticos sejam dispensados do logging, e em vez disso utilizar backups offline e intervenção humana.

Gerenciamento de transações em bancos de dados múltiplos

Lembre-se, pela seção "Bancos de dados distribuídos heterogêneos" do Capítulo 22, que o sistema de bancos de dados múltiplos cria a ilusão de integração lógica de banco de dados em um sistema de banco de dados heterogêneo, em que os sistemas de banco de dados locais podem empregar diferentes modelos de dados lógicos e linguagens de definição e manipulação de dados, e pode diferir em seus mecanismos de controle de concorrência e gerenciamento de transação.

Um sistema de bancos de dados múltiplos admite dois tipos de transações:

1. **Transações locais.** Essas transações são executadas por sistema de banco de dados local fora do controle do sistema de bancos de dados múltiplos.
2. **Transações globais.** Essas transações são executadas sob o controle do sistema de bancos de dados múltiplos.

O sistema de bancos de dados múltiplos está ciente do fato de que as transações locais podem ser executadas nos sites locais, mas não está ciente de quais transações específicas estão sendo executadas, ou de quais dados elas podem acessar.

Para garantir a autonomia local de cada sistema de banco de dados, é preciso que nenhuma mudança seja feita em seu software. Um sistema de banco de dados em um site, portanto, não é capaz de se comunicar diretamente com um em qualquer outro site para sincronizar a execução de uma transação global ativa em vários sites.

Como o sistema de bancos de dados múltiplos não tem controle sobre a execução de transações locais, cada sistema local precisa usar um esquema de controle de concorrência (por exemplo, bloqueio de duas fases ou timestamp) para garantir que seu escalonamento seja seriável. Além disso, no caso do bloqueio, o sistema local precisa ser capaz de se proteger contra a possibilidade de impasses locais.

A garantia de seriação local não é suficiente para garantir a seriação global. Como ilustração, considere duas transações globais T_1 e T_2, cada uma acessando e atualizando dois itens de dados, A e B, localizados nos sites S_1 e S_2, res-

pectivamente. Suponha que os escalonamentos locais sejam seriáveis. É possível ter uma situação em que no site S_1, T_2 venha após T_1, enquanto em S_2, T_1 venha após T_2, resultando em um escalonamento global não seriável. Na realidade, mesmo que não haja concorrência entre as transações globais (ou seja, uma transação global só é submetida depois que a anterior é confirmada ou abortada), a seriação local não é suficiente para garantir a seriação global (ver Exercício prático 25.7).

Dependendo da implementação dos sistemas de banco de dados locais, uma transação global pode não ser capaz de controlar o comportamento de bloqueio exato de suas subtransações locais. Assim, mesmo que todos os sistemas de banco de dados locais sigam o bloqueio em duas fases, pode ser possível apenas garantir que cada transação local siga as regras do protocolo. Por exemplo, um sistema de banco de dados local pode confirmar sua subtransação e liberar bloqueios, enquanto a subtransação em outro sistema local ainda esteja em execução. Se os sistemas locais permitirem o controle do comportamento de bloqueio e todos os sistemas seguirem o bloqueio em duas fases, então o sistema de bancos de dados múltiplos poderá garantir que as transações globais sejam bloqueadas em duas fases, e os pontos de bloqueio das transações em conflito definiriam então sua ordem de seriação global. Se diferentes sistemas locais seguirem diferentes mecanismos de controle de concorrência, essa classificação direta de controle global não funcionará.

Existem muitos protocolos para garantir a consistência apesar da execução concorrente de transações globais e locais em sistemas de bancos de dados múltiplos. Alguns são baseados na imposição de condições suficientes para garantir a seriação global. Outros garantem apenas uma forma de consistência mais fraca do que a seriação, mas alcançam essa consistência por meios menos restritivos. Consideramos um desses últimos esquemas: a *seriação em dois níveis*. A seção "Transações de longa duração" descreve outras técnicas para a consistência sem seriação; outras técnicas são citadas nas notas bibliográficas deste capítulo.

Um problema relacionado, nos sistemas de bancos de dados múltiplos, é o de commit atômico global. Se todos os sistemas locais seguirem o protocolo de commit em duas fases, esse protocolo poderá ser usado para conseguir a atomicidade global. Porém, os sistemas locais não projetados para fazer parte de um sistema distribuído podem não ser capazes de participar de tal protocolo. Mesmo que um sistema local seja capaz de admitir o commit em duas fases, a organização que possui o sistema pode não querer permitir a espera em casos em que ocorre bloqueio. Nesses casos, podem ser feitos ajustes para levar em consideração a falta de atomicidade em certos modos de falha. Outros aspectos relacionados aparecem na literatura (consulte as notas bibliográficas).

Seriação em dois níveis

A **seriação em dois níveis** (2LSR) garante a seriação em dois níveis do sistema:

- Cada sistema de banco de dados local garante a seriação local entre suas transações locais, incluindo aquelas que fazem parte de uma transação global.
- O sistema de bancos de dados múltiplos garante a seriação apenas entre as transações globais – *ignorando as ordenações induzidas pelas transações locais*.

Cada um desses níveis de seriação é simples de se impor. Os sistemas locais já oferecem garantias de seriação; assim, o primeiro requisito é fácil de se conseguir. O segundo requisito se aplica apenas a uma projeção do escalonamento global, em que as transações locais não aparecem. Assim, o sistema de bancos de dados múltiplos pode garantir o segundo requisito usando técnicas-padrão de controle de concorrência (a escolha da técnica exata não importa).

Os dois requisitos da 2LSR não são suficientes para garantir a seriação global. Porém, sob a técnica baseada em 2LSR, adotamos um requisito mais fraco que a seriação, chamado **exatidão forte**:

1. Preservação de consistência, especificada por um conjunto de restrições de consistência
2. Garantia de que o conjunto de itens de dados lidos por transação é consistente

Pode ser mostrado que certas restrições sobre o comportamento da transação, combinadas com 2LSR, são suficientes para garantir a exatidão forte (embora não necessariamente para garantir a seriação). Vamos listar várias dessas restrições.

Em cada um dos protocolos, distinguimos entre **dados locais** e **dados globais**. Os itens de dados globais pertencem a determinado site e estão sob o único controle desse site. Observe que não pode haver quaisquer restrições de consistência entre os itens de dados locais em sites distintos. Os itens de dados globais pertencem ao sistema de bancos de dados múltiplos e, embora possam ser armazenados em um site local, estão sob o controle do sistema de bancos de dados múltiplos.

O **protocolo de leitura global** permite que as transações leiam mas não atualizem itens de dados locais, enquanto não permite acesso total aos dados globais pelas transações locais. O protocolo de leitura global garante a exatidão forte se todas estas condições forem mantidas:

1. As transações locais acessam apenas itens de dados locais.
2. As transações globais podem acessar itens de dados globais e ler itens de dados locais (embora não devam escrever itens de dados locais).

3. Não existem restrições de consistência entre itens de dados locais e globais.

O **protocolo de leitura local** concede às transações locais o acesso de leitura aos dados globais, mas não permite o acesso total aos dados locais por transações globais. Nesse protocolo, temos de apresentar a noção de uma **dependência de valor**. Uma transação possui dependência de valor se o valor que ela escreve em um item de dados em um site depender de um valor que ela lê para um item de dados em outro site.

O protocolo de leitura local garante a exatidão forte se todas estas condições forem mantidas:

1. As transações locais podem acessar itens de dados locais e ler itens de dados globais armazenados no site (embora não devam escrever itens de dados globais).
2. As transações globais acessam apenas itens de dados globais.
3. Nenhuma transação pode ter dependência de valor.

O protocolo de **leitura-escrita global/leitura local** é o mais generoso em termos de acesso a dados dos protocolos que já consideramos. Ele permite que transações globais leiam e escrevam dados locais, e permite que transações locais leiam dados globais. Contudo, ele impõe tanto a condição de dependência de valor do protocolo de leitura local quanto a condição do protocolo de leitura global, de que não pode haver restrições de consistência entre dados locais e globais.

O protocolo de leitura-escrita global/leitura local garante a exatidão forte se todas estas condições forem mantidas:

1. As transações locais podem acessar itens de dados locais e ler itens de dados globais armazenados no site (embora não devam escrever itens de dados globais).
2. As transações globais podem acessar itens de dados globais e também itens de dados locais (ou seja, podem ler e escrever todos os dados).
3. Não existem restrições de consistência entre itens de dados locais e globais.
4. Nenhuma transação pode ter uma dependência de valor.

Garantia de seriação global

Os primeiros sistemas de bancos de dados múltiplos restringiam as transações globais a serem apenas lidas. Assim, eles evitavam a possibilidade de transações globais introduzindo inconsistência aos dados, mas não eram suficientemente restritivos para garantir a seriação global. É realmente possível obter esses escalonamentos globais e desenvolver um esquema para garantir a seriação global, e pediremos que você faça ambos no Exercício prático 25.8.

Existem diversos esquemas gerais para garantir a seriação global em um ambiente em que transações tanto de atuação quanto somente leitura podem ser executadas. Vários desses esquemas são baseados na idéia de um **ticket**. Um item de dados especial, chamado ticket, é criado em cada sistema de banco de dados local. Cada transação global que acessa dados em um site precisa escrever o ticket nesse site. Esse requisito garante que as transações globais entrarão em conflito direto em cada site que visitam. Além do mais, o gerenciador de transação global pode controlar a ordem em que as transações globais são seriadas, controlando a ordem em que os tickets são acessados. As referências a esses esquemas aparecem nas notas bibliográficas deste capítulo.

Se quisermos garantir a seriação global em um ambiente em que nenhum conflito local direto é gerado em cada site, algumas suposições precisam ser feitas sobre os escalonamentos permitidos pelo sistema de banco de dados local. Por exemplo, se os escalonamentos locais forem tais que a ordem de commit e a ordem de seriação sejam sempre idênticas, podemos garantir a seriação controlando apenas a ordem em que as transações são confirmadas.

O problema com esquemas que garantem a seriação global é que eles podem restringir a concorrência indevidamente. Eles têm uma probabilidade maior de fazer isso porque a maioria das transações submete instruções SQL ao sistema de banco de dados básico, em vez de submeter etapas individuais de **read**, **write**, **commit** e **abort**. Embora não sendo possível garantir a seriação global sob essa suposição, o nível de concorrência pode ser tal que outros esquemas sejam alternativas atraentes, como a técnica de seriação em dois níveis, discutida na seção "Seriação em dois níveis".

Resumo

- Fluxos de trabalho são atividades que envolvem a execução coordenada de várias tarefas realizadas por diferentes entidades de processamento. Eles existem não apenas em aplicações de computador, mas também em quase todas as atividades organizacionais. Com o crescimento das redes e a existência de múltiplos sistemas de banco de dados autônomos, os fluxos de trabalho oferecem uma maneira conveniente de executar tarefas que envolvem múltiplos sistemas.
- Embora os requisitos transacionais ACID normais sejam muito fortes ou impossíveis de implementar para tais aplicações de fluxo de trabalho, os fluxos de trabalho precisam satisfazer um conjunto limitado de propriedа-

des transacionais que garantem que um processo não seja deixado em um estado inconsistente.

- Os monitores de processamento de transação foram desenvolvidos inicialmente como servidores multithreaded que poderiam atender a grandes quantidades de terminais a partir de um único processo. Eles evoluíram desde então, e hoje oferecem a infra-estrutura para a montagem e administração de sistemas complexos de processamento de transação, que possuem uma grande quantidade de clientes e vários servidores. Eles oferecem serviços como enfileiramento durável de solicitações do cliente e respostas do servidor, roteamento de mensagens do cliente aos servidores, mensagens persistentes, balanceamento de carga e coordenação de commit de duas fases quando as transações acessam vários servidores.
- Sistemas de e-commerce (comércio eletrônico) se tornaram uma parte essencial do comércio. Existem várias questões relacionadas a banco de dados nos sistemas de e-commerce. O gerenciamento de catálogo, especialmente a personalização do catálogo, é feito com bancos de dados. Os mercados eletrônicos ajudam na definição de preços de produtos por meio de leilões, leilões reversos ou permutas. Os sistemas de banco de dados de alto desempenho são necessários para lidar com esse tipo de comércio. Os pedidos são estabelecidos por sistemas de pagamento eletrônico, que também precisam de sistemas de banco de dados de alto desempenho para lidar com taxas de transação muito altas.
- Grandes memórias principais são exploradas em certos sistemas para conseguir um throughput de sistema muito alto. Nesses sistemas, o logging é um gargalo. Sob o conceito de commit em grupo, o número de saídas para o armazenamento estável pode ser reduzido, liberando assim esse gargalo.
- O gerenciamento eficiente de transações interativas de longa duração é mais complexo, devido às esperas de longa duração e devido à possibilidade de abortos. Como as técnicas de controle de concorrência usadas no Capítulo 16 utilizam esperas, abortos ou ambos, técnicas alternativas precisam ser consideradas. Essas técnicas precisam garantir a exatidão sem exigir a seriação.
- Uma transação de longa duração é representada como uma transação aninhada com as operações de banco de dados atômicas no nível mais baixo. Se uma transação falhar, somente as transações ativas de curta duração abortam. As transações ativas de longa duração retomam quando qualquer transação de curta duração tiver sido recuperada. Uma transação de compensação é necessária para desfazer as atualizações de transações aninhadas que foram confirmadas, se a transação de nível mais externo falhar.
- Em sistemas com restrições de tempo real, a exatidão da execução envolve não apenas a consistência do banco de dados, mas também a satisfação do prazo. A grande variação de tempos de execução para operações de leitura e escrita complica o problema de gerenciamento de transação para sistemas restritos no tempo.
- Um sistema com múltiplos bancos de dados oferece um ambiente em que novas aplicações de banco de dados podem acessar dados de uma série de bancos de dados preexistentes localizados em vários ambientes de hardware e software heterogêneos.

  Os sistemas de banco de dados locais podem empregar diferentes modelos lógicos e linguagens de definição e manipulação de dados, e podem diferir em seus mecanismos de controle de concorrência e gerenciamento de transação. Um sistema de múltiplos bancos de dados cria a ilusão de integração lógica de banco de dados, sem exigir a integração física do banco de dados.

Termos de revisão

- Monitor de PT
- Arquiteturas de monitor de PT
  - Processo por cliente
  - Único servidor
  - Muitos servidores, único roteador
  - Muitos servidores, muitos roteadores
- Multitarefa
- Troca de contexto
- Servidor multithreaded
- Gerenciador de fila
- Coordenação de aplicações
  - Gerenciador de recursos
  - Chamada de procedimento remoto (RPC – Remote Procedure Call)
- Fluxos de trabalho transacionais
  - Tarefa
  - Entidade de processamento
  - Especificação de fluxo de trabalho
  - Execução de fluxo de trabalho
- Estado do fluxo de trabalho
  - Estados de execução
  - Valores de saída
  - Variáveis externas
- Atomicidade de falha do fluxo de trabalho
- Estados de término do fluxo de trabalho
  - Aceitáveis
  - Não aceitáveis
  - Confirmados
  - Abortados
- Recuperação de fluxo de trabalho
- Sistema de gerenciamento de fluxo de trabalho

- Arquiteturas do sistema de gerenciamento de fluxo de trabalho
  - Centralizada
  - Parcialmente distribuída
  - Totalmente distribuída
- E-commerce
- E-catalogs
- Mercados
  - Leilões
  - Leilões reversos
  - Permuta
- Estabelecimento de pedido
- Certificados digitais
- Bancos de dados na memória principal
- Commit em grupo
- Sistemas de tempo real
- Prazos
  - Prazo rígido
  - Prazo firme
  - Prazo flexível
- Bancos de dados de tempo real
- Transações de longa duração
- Exposição de dados não confirmados
- Execuções não seriáveis
- Transações aninhadas
- Transações multinível
- Saga
- Transações de compensação
- Logging lógico
- Sistemas de bancos de dados múltiplos
- Autonomia
- Transações locais
- Transações globais
- Seriação em dois níveis (2LSR)
- Exatidão forte
- Dados locais
- Dados globais
- Protocolos
  - Leitura global
  - Leitura local
  - Dependência de valor
  - Leitura-escrita global/leitura local
- Garantia de seriação global
- Ticket

Exercícios práticos

25.1 Assim como os sistemas de banco de dados, os sistemas de fluxo de trabalho também exigem gerenciamento de concorrência e recuperação. Liste três motivos pelos quais não podemos simplesmente aplicar um sistema de banco de dados relacional usando 2PL, logging de undo físico e 2PC.

25.2 Considere um sistema de banco de dados da memória principal recuperando-se de uma falha do sistema. Explique os méritos relativos de
- Carga do banco de dados inteiro de volta à memória principal antes de retomar o processamento da transação
- Carga de dados conforme são solicitados pelas transações

25.3 Um sistema de transação de alto desempenho é necessariamente um sistema de tempo real? Por que ou por que não?

25.4 Explique por que pode não ser prático exigir seriação para transações de longa duração.

25.5 Considere um processo multithreaded que entrega mensagens de uma fila durável de mensagens persistentes. Diferentes threads podem ser executados simultaneamente, tentando oferecer diferentes mensagens. No caso de uma falha de remessa, a mensagem precisa ser restaurada na fila. Modele as ações que cada thread executa como uma transação multinível, de modo que os bloqueios sobre a fila não precisem ser mantidos até que uma mensagem seja entregue.

25.6 Discuta as modificações que precisam ser feitas em cada um dos esquemas de recuperação abordados no Capítulo 17 se permitirmos transações aninhadas. Além disso, explique quaisquer diferenças resultantes se permitirmos transações multinível.

25.7 Considere um sistema de bancos de dados múltiplos em que é garantido que no máximo uma transação global esteja ativa a qualquer momento, e que cada site local garanta seriação local.

a. Sugira maneiras como o sistema de banco de dados múltiplos pode garantir que existe no máximo uma transação global ativa a qualquer momento.

b. Mostre, por meio de exemplo, que é possível que um escalonamento global não seriável aconteça apesar das suposições.

25.8 Considere um sistema de bancos de dados múltiplos em que cada site local garante a seriação local e todas as transações globais sejam somente leitura.

a. Mostre, por meio de exemplo, que as execuções não seriáveis podem acontecer em tal sistema.

b. Mostre como você poderia usar um esquema de ticket para garantir a seriação global.

Exercícios

25.9 Explique como um monitor de PT gerencia recursos de memória e processador de forma mais eficiente que um sistema operacional típico.

25.10 Compare os recursos de monitor de PT com aqueles oferecidos pelos servidores Web que dão suporte a servlets (esses servidores foram apelidados de *PT-lite*).

25.11 Considere o processo de admissão de novos alunos em sua universidade (ou novos funcionários em sua organização).

a. Crie uma imagem de alto nível do fluxo de trabalho começando com o procedimento de pedido de ingresso do aluno.

b. Indique os estados de término aceitáveis e quais etapas envolvem intervenção humana.

c. Indique possíveis erros (incluindo expiração de prazo) e como eles são tratados.

d. Estude quanto do fluxo de trabalho foi automatizado na sua universidade.

25.12 Responda as seguintes perguntas considerando os sistemas de pagamento eletrônico.

a. Explique por que as transações eletrônicas executadas por números de cartão de crédito são inseguras.

b. Uma alternativa é ter um gateway de pagamento eletrônico mantido pela companhia de cartão de crédito, e o site recebendo pagamento redirecionar clientes ao site do gateway para fazer o pagamento.

i. Explique quais benefícios esse sistema oferece se o gateway não autenticar o usuário.

ii. Explique que outros benefícios são oferecidos se o gateway tiver um mecanismo para autenticar o usuário.

c. Algumas empresas de cartão de crédito oferecem um número de cartão de crédito para uso único como um método mais seguro de pagamento eletrônico. Os clientes se conectam ao site das companhias de cartão de crédito para obter o número para uso único. Explique que benefício esse sistema oferece, em comparação com o uso dos números normais de cartão de crédito. Além disso, explique seus benefícios e desvantagens em comparação com os gateways de pagamento eletrônico com autenticação.

d. Algum desses sistemas garante a mesma privacidade que existe quando os pagamentos são feitos em dinheiro? Explique sua resposta.

25.13 Se o banco de dados inteiro couber na memória principal, ainda precisamos de um sistema de banco de dados para gerenciar os dados? Explique sua resposta.

25.14 Nas técnica de commit em grupo, quantas transações devem fazer parte de um grupo? Explique a sua resposta.

25.15 Em um sistema de banco de dados usando logging de escrita antecipada, qual é a quantidade de acessos ao disco, no pior caso, exigida para ler um item de dados de uma página de disco especificada? Explique por que isso apresenta um problema para os projetistas de sistemas de banco de dados de tempo real. Dica: considere o caso em que o buffer de disco está cheio.

25.16 Qual é a finalidade das transações de compensação? Apresente dois exemplos de seu uso.

25.17 Explique as conexões entre um fluxo de trabalho e uma transação de longa duração.

Notas bibliográficas

Gray e Edwards [1995] oferecem uma visão geral das arquiteturas do monitor de PT; Gray e Reuter [1993] oferecem um livro-texto detalhado (e excelente) dos sistemas de processamento de transação, incluindo capítulos sobre monitores de PT. X/Open [1991] define a interface X/Open XA. O processamento de transação na Tuxedo é descrito em Huffman [1993]. Wipfler [1987] é um dos vários textos sobre desenvolvimento de aplicação usando CICS.

Fischer [2001] é um manual sobre sistemas de fluxo de trabalho. Hollingsworth [1994] apresenta um modelo de referência para fluxos de trabalho, proposto pela Workflow Management Coalition. Uma revisão do modelo, incluindo suas conexões com a *modelagem do processo comercial*, é apresentada por Hollingsworth [2004], como parte de um manual sobre fluxos de trabalho (Fischer [2004]). O site da coalizão é `www.wfmc.org`. Nossa descrição de fluxos de trabalho segue o modelo de Rusinkiewicz e Sheth [1995].

Loeb [1998] oferece uma descrição detalhada das transações eletrônicas seguras.

Garcia-Molina e Salem [1992] oferecem uma visão geral dos bancos de dados de memória principal. Jagadish *et al.* [1993] descrevem um algoritmo de recuperação projetado para bancos de dados da memória principal. Um gerenciador de armazenamento para bancos de dados da memória principal é descrito em Jagadish *et al.* [1994].

Os bancos de dados de tempo real são discutidos por Lam e Kuo [2001]. Aplicações de processamento de dados em tempo real incluem TelegraphCQ (Chandrasekaran *et al.* [2003]). O controle de concorrência e o escalonamento nos bancos de dados de tempo real são discutidos por Haritsa *et al.* [1990], Hong *et al.* [1993] e Pang *et al.* [1995]. Ozsoyoglu e Snodgrass [1995] é um trabalho de pesquisa sobre bancos de dados de tempo real e temporais.

Transações aninhadas e multinível são apresentadas por Lynch [1983], Moss [1985], Lynch e Merritt [1986], Fekete *et al.* [1990], Weikum [1991], Korth e Speegle [1994] e Pu *et al.* [1988]. Aspectos teóricos das transações multiní-

vel são apresentados em Lynch et al. [1988] e Weihl e Liskov [1990].

Diversos modelos de transação estendida foram definidos, incluindo Sagas (Garcia-Molina e Salem [1987]), o modelo ConTract (Wachter e Reuter [1992]) e ARIES (Mohan *et al.* [1992] e Rothermel e Mohan [1989]). A divisão de transações para conseguir maior desempenho é tratada em Shasha *et al.* [1995]. A recuperação nos sistemas de transação aninhados é discutida por Moss [1987], Haerder e Rothermel [1987] e Rothermel e Mohan [1989].

O processamento de transação para transações de longa duração é considerado por Weikum e Schek [1984], Haerder e Rothermel [1987], Weikum *et al.* [1990] e Korth *et al.* [1990]. Salem *et al.* [1994] apresentam uma extensão do 2PL para transações de longa duração, permitindo a liberação antecipada de bloqueios sob certas circunstâncias.

O processamento de transações nos sistemas de bancos de dados múltiplos é discutido em Mehrotra *et al.* [2001]. O esquema de ticket é apresentado em Georgakopoulos *et al.* [1994]. 2LSR é apresentado em Mehrotra *et al.* [1991].

PARTE 9

Estudos de caso

Esta parte descreve como diferentes sistemas de banco de dados integram os diversos conceitos descritos anteriormente no livro. Começamos abordando um sistema de banco de dados de fonte aberto muito usado, PostgreSQL, no Capítulo 26. Três sistemas de banco de dados comerciais muito usados – IBM DB2, Oracle e Microsoft SQL Server – são explicados nos Capítulos 27, 28 e 29.

Cada um desses capítulos destaca recursos exclusivos de cada sistema de banco de dados: ferramentas, variações e extensões da SQL, além de arquitetura do sistema, incluindo organização de armazenamento, processamento de consulta, controle de concorrência e recuperação, e também replicação.

Os capítulos abordam apenas aspectos básicos dos produtos de banco de dados que eles descrevem, e por isso não devem ser considerados como uma cobertura abrangente do produto. Além do mais, como os produtos são melhorados regularmente, seus detalhes podem mudar. Ao usar determinada versão do produto, certifique-se de consultar os manuais do usuário em busca de detalhes específicos.

Lembre-se de que os capítulos desta parte normalmente utilizam a terminologia industrial, em vez da acadêmica. Por exemplo, eles utilizam tabela em vez de relação, linha em vez de tupla e coluna em vez de atributo.

CAPÍTULO 26

PostgreSQL

Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder
CMU

PostgreSQL é um sistema de gerenciamento de banco de dados objeto relacional de fonte aberto. Ele é descendente de um dos sistemas mais antigos, o sistema Postgres, desenvolvido pelo Professor Michael Stonebraker da Universidade da Califórnia em Berkeley. O nome "Postgres" é derivado do nome de um sistema de banco de dados relacional pioneiro, Ingres, também desenvolvido por Stonebraker em Berkeley. Atualmente, PostgreSQL admite SQL92 e SQL:1999 e oferece recursos como consultas complexas, chaves estrangeiras, triggers, views, integridade transacional e controle de concorrência de múltiplas versões. Além disso, os usuários podem estender o PostgreSQL com novos tipos de dados, funções, operadores ou métodos de índice. PostgreSQL funciona com diversas linguagens de programação (incluindo C, C++, Java, Perl, Tcl e Python). Talvez o ponto mais forte do PostgreSQL seja que ele, junto com MySQL, sejam os dois sistemas de banco de dados relacional de fonte aberto mais usados. A licença do PostgreSQL é a licença BSD, que dá permissão a qualquer um para usar, modificar e distribuir o código e a documentação do PostgreSQL para qualquer finalidade gratuitamente.

Introdução

No decorrer de mais de uma década, o PostgreSQL passou por várias versões importantes. O primeiro sistema protótipo, denominado Postgres, foi demonstrado na conferência ACM SIGMOD de 1988. A versão 1 foi distribuída aos usuários em 1989. Depois que as versões subseqüentes acrescentaram um novo sistema de regras, suporte para vários gerenciadores de armazenamento e o executor de consulta melhorado, os desenvolvedores do sistema enfocaram a portabilidade e o desempenho até 1994, quando foi acrescentado o interpretador da linguagem SQL. Sob um novo nome, Postgres95, o sistema foi lançado na Web e mais tarde comercializado pela Illustra Information Technologies (que depois se juntou à Informix, agora pertencente à IBM). Em 1996, o nome Postgres95 foi substituído por PostgreSQL, para refletir o relacionamento entre o Postgres original e as versões mais recentes com capacidade para SQL.

PostgreSQL é executado em praticamente todos os sistemas operacionais tipo Unix, incluindo Linux e Apple Macintosh OS X. PostgreSQL pode ser executado no Microsoft Windows, no ambiente Cygwin, que oferece emulação Linux no Windows. A versão 8.0, lançada em janeiro de 2005, oferece suporte nativo para Microsoft Windows.

Hoje, PostgreSQL é usado para implementar várias aplicações de pesquisa e produção diferentes (como o projeto de computação científica Sequoia 2000) e é uma ferramenta educacional em várias universidades. O sistema continua a evoluir por meio das contribuições de uma comunidade de cerca de 1.000 desenvolvedores. Neste capítulo, explicamos como funciona o PostgreSQL, começando pelas interfaces com o usuário e linguagens, e continuando no núcleo do sistema (as estruturas de dados e mecanismo de controle de concorrência).

Interfaces com o usuário

A distribuição padrão do PostgreSQL vem com ferramentas da linha de comandos para administrar o banco de dados. Porém, existe uma grande quantidade de ferramentas gráficas de administração e projeto de fonte aberto que dão suporte para PostgreSQL. PostgreSQL oferece um abrangente conjunto de interfaces de programação.

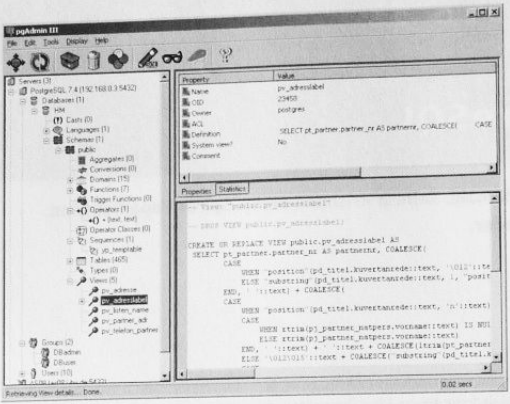


Figura 26.1 Uma GUI de administração de banco de dados de fonte aberto.

Interfaces de terminal interativo

Como a maioria dos sistemas de banco de dados, PostgreSQL oferece ferramentas da linha de comandos para administração de banco de dados. O principal cliente de terminal interativo é psql, que é baseado no shell Unix e permite a execução de comandos SQL no servidor, além de várias outras operações (como cópia no cliente). Alguns de seus recursos são:

- **Variáveis.** `psql` oferece recursos de substituição de variáveis, de modo semelhante aos shells de comando do Unix.
- **Interpolação de SQL.** O usuário pode substituir ("interpolar") variáveis psql em instruções SQL normais colocando um ponto-e-vírgula na frente do nome da variável.
- **Edição da linha de comandos.** `psql` utiliza a biblioteca GNU readline para realizar a edição de linha conveniente, com suporte para conclusão com tab.

PostgreSQL também possui `pgtksh` e `pgtclsh`, que são versões dos shells Tk e Tcl (`wish`), que tradicionalmente incluem vínculos do PostgreSQL. Tcl/Tk é uma linguagem de scripting flexível, normalmente usada para o protótipo rápido.

Interfaces gráficas

A distribuição padrão do PostgreSQL não contém ferramentas gráficas. Porém, existem diversas ferramentas de interface gráfica com o usuário, e os usuários podem escolher dentre alternativas comerciais e de fonte aberto. Muitas destas passam por ciclos de lançamento rápidos; a lista a seguir reflete o estado das coisas no momento em que escrevemos.

Existem ferramentas gráficas para administração, incluindo pgAccess e pgAdmin, sendo que esta última aparece na Figura 26.1. Ferramentas para projeto de banco de dados incluem TORA e Data Architect, sendo que esta última aparece na Figura 26.2.

PostgreSQL funciona com várias ferramentas comerciais de projeto de formulários e geração de relatórios. As alternativas de fonte aberta incluem Rekall (que aparece nas Figuras 26.3 e 26.4), GNU Report Generator e um pacote de ferramentas mais abrangente, GNU Enterprise.

Interfaces de linguagem de programação

PostgreSQL oferece interfaces nativas para ODBC e JDBC, além de vínculos para a maioria das linguagens de programação, incluindo C, C++, PHP, Perl, Tcl/Tk, ECPG, Python e Ruby.

A API C para o PostgreSQL é libpq, que também é o mecanismo básico para a maioria dos vínculos de linguagem de programação (assim, todos os seus recursos também estão disponíveis nas outras linguagens aceitas). A biblioteca libpq admite execução síncrona e assíncrona de comandos SQL e instruções preparadas. Ela é reentrante e segura para

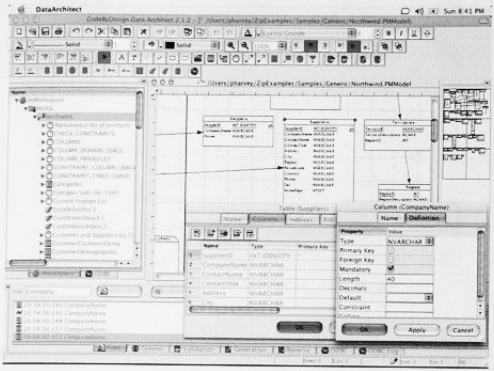


Figura 26.2 Uma GUI de projeto de banco de dados para múltiplas plataformas.

thread. Ela usa variáveis de ambiente para certos parâmetros e um arquivo de senhas opcional para conexões que exigem autenticação.

Variações e extensões da SQL

PostgreSQL é compatível com ANSI SQL. Ele admite quase todos os recursos da SQL92 em nível de entrada (o que a maioria dos fornecedores de sistema de banco de dados relacional quer dizer com conformidade SQL92) e muitos dos recursos de nível intermediário e completo. Finalmente, ele admite vários dos recursos da SQL:1999 (incluindo a maioria dos recursos objeto relacional descritos no Capítulo 9); de fato, alguns destes (como arrays, funções e herança) foram iniciados pelo PostgreSQL ou seus ancestrais. Ele não possui os recursos OLAP (principalmente, **cube** e **rol-**

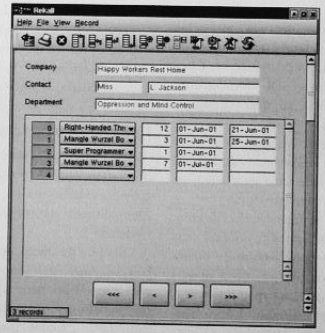


Figura 26.3 Rekall: GUI de projeto de formulário.

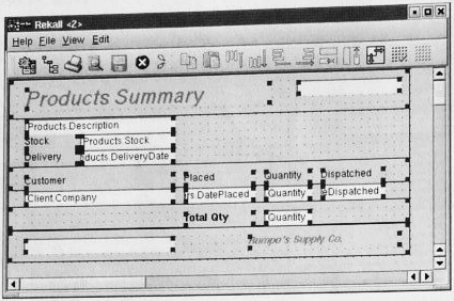


Figura 26.4 Rekall: GUI de projeto de relatório.

lup), mas os dados do PostgreSQL podem ser facilmente carregados em servidores OLAP externos de fonte aberto (como Mondrian), além de produtos comerciais.

Tipos PostgreSQL

PostgreSQL possui suporte para vários tipos fora do padrão, úteis para domínios de aplicação específicos. Além do mais, os usuários podem definir novos tipos com o comando **create type**. Isso inclui novos tipos básicos de baixo nível, normalmente escritos em C (ver seção "Tipos").

O sistema de tipos do PostgreSQL

Os tipos PostgreSQL podem ser das seguintes categorias:

- **Tipos básicos.** Os tipos básicos também são conhecidos como **tipos de dados abstratos**; ou seja, os módulos que encapsulam estado e um conjunto de operações. Estes são implementados abaixo do nível SQL, normalmente em uma linguagem como C (ver seção "Tipos"). Alguns exemplos são **int4** (já incluídos no PostgreSQL) ou **complex** (incluído como um tipo de extensão opcional). Um tipo básico em PostgreSQL é acompanhado automaticamente por um tipo array que pode armazenar arrays de tamanho variável do tipo básico em particular.
- **Tipos compostos.** Estes correspondem a linhas de tabela; ou seja, eles são uma lista de nomes de campo e seus respectivos tipos básicos. Um tipo composto é criado implicitamente sempre que uma tabela é criada, mas os usuários também podem construí-los explicitamente.
- **Domínios.** Estes são muito semelhantes aos tipos básicos (e normalmente os dois são intercambiáveis), mas podem ter restrições sobre valores permitidos.
- **Pseudotipos.** Atualmente, PostgreSQL admite os seguintes pseudotipos: *any*, *anyarray*, *anyelement*, *cstring*, *internal*, *language-handler*, *record*, *trigger* e *void*. Estes não podem ser usados em tipos compostos (e por isso não podem ser usados para colunas de tabela), mas podem ser usados como tipos de argumento e retorno das funções definidas pelo usuário.
- **Tipos polimórficos.** Os dois pseudotipos *anyelement* e *anyarray* são coletivamente conhecidos como **polimórficos**. As funções com argumentos desses tipos (correspondentemente chamadas **funções polimórficas**) podem operar sobre qualquer tipo real. PostgreSQL possui um esquema de resolução de tipo simples, que exige que: (1) em qualquer invocação específica de uma função polimórfica, todas as ocorrências de um tipo polimórfico estejam ligadas ao mesmo tipo real (ou seja, uma função definida como *f(anyelement, anyelement)* só pode operar sobre pares do mesmo tipo real), e (2) se o tipo de retorno for polimórfico, então pelo menos um dos argumentos precisa ser do mesmo tipo polimórfico.

Tipos fora do padrão

Os tipos descritos nesta seção estão incluídos na distribuição padrão. Além do mais, graças à natureza aberta do PostgreSQL, existem vários tipos de extensão contribuídos, como números complexos e ISBN/ISSNs (ver seção "Extensibilidade").

Os tipos de dados geométricos (*point*, *line*, *lseg*, *box*, *polygon*, *path*, *circle*) são usados em sistemas de informações geográficas para representar objetos espaciais bidimensionais, como pontos, segmentos de linha, polígonos, caminhos e círculos. Diversas funções e operadores estão disponíveis no PostgreSQL para realizar diversas operações

geométricas, como escala, tradução, rotação e determinação de interseções. Além do mais, PostgreSQL admite indexação desses tipos usando árvores R (seções "Árvores R" do Capítulo 24 e "Tipos de índice").

PostgreSQL oferece tipos de dados para armazenar endereços de rede. Esses tipos de dados permitem que aplicações de gerenciamento de rede utilizem um banco de dados PostgreSQL como seu depósito de dados. Para os que estão acostumados com redes de computador, oferecemos um rápido resumo desse recurso aqui. Existem tipos separados para endereços IPv4, IPv6 e Media Access Control (MAC) (*cidr*, *inet* e *macaddr*, respectivamente). Tanto os tipos *inet* quanto *cidr* podem armazenar endereços IPv4 e IPv6, com máscaras de sub-rede opcionais. Sua principal diferença está na formatação de entrada/saída, além da restrição de que os endereços CIDR (Classless Internet Domain Routing) não aceitam valores com bits diferentes de zero à direita da máscara de rede. O tipo *macaddr* é usado para armazenar endereços MAC (normalmente, endereços de hardware de placa Ethernet). PostgreSQL admite indexação e classificação sobre esses tipos, além de um conjunto de operações (incluindo teste de sub-rede e mapeamento de endereços MAC a nomes de fabricante de hardware). Além do mais, esses tipos oferecem verificação em nível de entrada. Assim, eles são preferíveis aos campos de texto puro.

O tipo *bit* do PostgreSQL pode armazenar seqüências de 1s e 0s de tamanho fixo e variável. PostgreSQL admite operadores lógicos de bit e funções de manipulação de strings.

Regras e outros recursos de banco de dados ativo

PostgreSQL admite restrições e triggers SQL (e procedimentos armazenados; ver próxima seção). Além do mais, ele possui regras de reescrita de consulta que podem ser declaradas no servidor.

PostgreSQL permite restrições de integridade, restrições not-null e restrições de chave primária e chave estrangeira (com restrição e propagação de exclusões).

Como muitos outros sistemas de bancos de dados relacionais, PostgreSQL aceita triggers, que são úteis para restrições não triviais e verificação e imposição de consistência. As funções de trigger podem ser escritas em uma linguagem de procedimento como PL/pgSQL (ver seção "Linguagens procedurais") ou em C, mas não em SQL pura. As triggers podem ser executadas antes ou depois de operações **insert**, **update** ou **delete**, e uma vez por linha modificada ou uma vez por instrução SQL.

O sistema de regras do PostgreSQL permite que os usuários definam regras de reescrita de consulta no servidor de banco de dados. Diferente dos procedimentos armazenados e triggers, o sistema de regras intervém entre o analisador de consulta e o planejador, e modifica consultas com base no conjunto de regras. Depois que a árvore de consulta original tiver sido transformada em uma ou mais árvores, elas são passadas para o planejador de consulta. Assim, o planejador tem toda a informação necessária (tabelas a serem varridas, relacionamentos entre elas, qualificações, informações de junção e assim por diante) e podem vir com um plano de execução eficiente, mesmo quando são envolvidas regras complexas.

A sintaxe geral para a declaração de regras é:

```
create rule nome_regra as
    on [ select | insert | update | delete ]
    to tabela [ where qualificação_regra ]
    do [ instead ] [ nothing | comando | ( comando ;
    comando ... ) ]
```

O restante desta seção contém exemplos que ilustram as capacidades do sistema de regras. Outros detalhes de como as regras são combinadas com árvores de consulta, e como estas mais tarde são transformadas, poderão ser encontrados na documentação do PostgreSQL (ver as notas bibliográficas). O sistema de regras é implementado na fase de reescrita do processamento da consulta e explicado na seção "Reescrita de consulta".

Primeiro, PostgreSQL usa regras para implementar views. Uma definição de view como esta:

```
create view minhaview as select * from minhatabela;
```

é convertida para a seguinte definição de regra:

```
create table minhaview (mesma lista de colunas de
        minhatabela);
create rule return as on select to minhaview do instead
        select * from minhatabela;
```

As consultas sobre *minhaview* são transformadas antes da execução em consultas sobre a tabela básica *minhatabela*. A sintaxe de **create view** é considerada uma melhor forma de programação nesse caso, pois é mais concisa e também evita a criação de views que referenciam uma à outra (que é possível se as regras forem declaradas descuidadamente, resultando em erros de runtime potencialmente confusos). Porém, podem ser usadas regras para definir explicitamente ações de atualização sobre views (instruções **create view** não permitem isso).

Como outro exemplo, considere o caso em que o usuário deseja auditar atualizações de tabela. Algo assim poderia ser conseguido por uma regra como:

```
create rule auditar_salário as on update to funcionário
    where new.salário < > old.salário
    do insert into auditar_salário
    values (current-timestamp, current-user,
            new.nome_func, old.salário, new.salário);
```

Finalmente, vamos mostrar uma regra de insert/update um pouco mais complicada. Suponha que os aumentos de salário pendentes sejam armazenados em uma tabela *aumentos_salário*(*nome_func*,*aumento*). Podemos declarar uma tabela "fictícia" *aumentos_aprovados* com os mesmos campos e depois definir a seguinte regra:

```
create rule inserir_aumentos_aprovados
    as on insert to aumentos_aprovados
    do instead
    update funcionário
        set salário = salário + new.aumento
        where nome_func = new.nome_func;
```

Então, a seguinte consulta:

```
insert into aumentos_aprovados select * from
            aumentos_salário;
```

atualizará todos os salários na tabela *funcionário* de uma só vez.

O sistema de regras PostgreSQL pode ser usado para implementar a maioria dos triggers. Alguns tipos de restrições (especialmente as chaves estrangeiras) não podem ser implementados por regras. Além disso, se a violação de uma restrição gerar uma mensagem de erro (em vez de descartar silenciosamente os valores inválidos, declarando uma regra "...**do instead nothing**"), então as triggers precisam ser usadas. As triggers não podem ser usadas para ações **update** ou **delete** sobre views. Como não existem dados reais em uma relação de view, a trigger nunca seria chamada.

Finalmente, uma trigger é disparada uma vez para cada linha afetada. Uma regra, por outro lado, manipula a árvore de consultas antes do planejamento da consulta. Assim, se uma instrução afetar muitas linhas, uma regra será muito mais eficiente do que uma trigger.

A implementação de trigger e restrições em PostgreSQL é esboçada rapidamente na seção "Triggers e restrições".

Extensibilidade

Como na maioria dos sistemas de bancos de dados relacionais, PostgreSQL armazena informações sobre bancos de dados, tabelas, colunas e assim por diante naquilo que normalmente é conhecido como **catálogos do sistema**, que aparecem para o usuário como tabelas normais. Outros sistemas de banco de dados relacionais normalmente são estendidos pela mudança de procedimentos codificados diretamente no código-fonte ou pela carga de módulos de extensão especiais escritos pelo fornecedor.

Porém, ao contrário da maioria dos sistemas de banco de dados relacionais, PostgreSQL vai um passo adiante e armazena muito mais informações em seus catálogos: não apenas informações sobre tabelas e colunas, mas também informações sobre tipos de dados, funções, métodos de acesso e assim por diante. Portanto, PostgreSQL é fácil para os usuários estenderem e facilita o protótipo rápido de novas aplicações e estruturas de armazenamento. PostgreSQL também pode incorporar código escrito pelo usuário no servidor, por meio de carregamento dinâmico de objetos compartilhados. Isso provê uma técnica alternativa para a escrita de extensões que possam ser usadas quando as extensões baseadas em catálogo não forem suficientes.

Além do mais, o módulo contrib da distribuição do PostgreSQL inclui diversas funções do usuário (por exemplo, iteradores de array, combinação de string difusa, funções criptográficas), tipos básicos (por exemplo, senhas criptografadas, ISBN/ISSNs, cubos de *n* dimensões) e extensões de índice (por exemplo, árvores RD, indexação de texto completo). Graças à natureza aberta do PostgreSQL, há uma grande comunidade de profissionais e entusiastas do PostgreSQL que também estendem bastante o PostgreSQL quase diariamente. Os tipos de extensão são idênticos em funcionalidade aos tipos embutidos (ver também a seção "Tipos fora do padrão"); estes últimos simplesmente já estão vinculados ao servidor e previamente registrados no catálogo do sistema. De modo semelhante, essa é a única diferença entre funções embutidas e de extensão.

Tipos

PostgreSQL permite que os usuários definam tipos compostos, além de estenderem os tipos básicos disponíveis.

Uma definição de tipo composto é semelhante a uma definição de tabela (na verdade, esta última implicitamente realiza a primeira). Tipos compostos independentes normalmente são úteis como argumentos de função. Por exemplo, a definição

```
create type t_cidade as (nome varchar(80), estado char(2))
```

permite que as funções aceitem e retornem tuplas *t_cidade*, mesmo que não haja tabela que contenha explicitamente as linhas desse tipo.

A inclusão de tipos básicos ao PostgreSQL é simples; um exemplo pode ser encontrado em complex.sql e complex.c nos tutoriais da distribuição do PostgreSQL. O tipo básico pode ser declarado em C, por exemplo:

```
typedef struct Complex {
  double x;
  double y;
} Complex;
```

Então, o usuário precisa definir funções para ler e escrever valores do novo tipo no formato de texto (ver seção "Funções"). Depois disso, o novo tipo pode ser registrado por meio da instrução:

```
create type complexo (
    internallength = 16,
    input = complexo_in,
    output = complexo_out,
    alignment = double
);
```

considerando que as funções de E/S de texto foram registradas como *complexo_in* e *complexo_out*.

O usuário também tem a opção de definir funções de E/S binárias (para o dumping de dados mais eficiente). Os tipos de extensão podem ser usados como os tipos básicos existentes do PostgreSQL. De fato, sua única diferença é que os tipos de extensão são carregados e vinculados dinamicamente no servidor. Além do mais, os índices podem ser facilmente estendidos para lidar com novos tipos básicos; ver a seção "Extensibilidade".

Funções

PostgreSQL permite que os usuários definam funções que são armazenadas e executadas no servidor. PostgreSQL também admite sobrecarga de função (ou seja, funções que podem ser declaradas com o mesmo nome, mas com argumentos de tipos diferentes). As funções podem ser escritas como instruções SQL puras. Além disso, várias linguagens procedurais são aceitas (explicadas na seção "Linguagens procedurais"). Finalmente, PostgreSQL possui uma API para acrescentar funções escritas em C (que explicamos ainda nesta seção).

Funções definidas pelo usuário podem ser escritas em C (ou em uma linguagem com convenções de chamada compatíveis, como C++). As convenções de codificação reais são basicamente as mesmas para as funções carregadas dinamicamente, definidas pelo usuário, além de funções internas (que são vinculadas estaticamente ao servidor). Logo, a biblioteca de funções internas padrão é uma rica fonte de exemplos de codificação para funções C definidas pelo usuário. Quando a biblioteca compartilhada que contém a função tiver sido criada, uma declaração como a seguinte a registra no servidor:

```
create function complexo_out(complexo)
    returns cstring
    as 'arquivo_objeto_compartilhado'
    language C immutable strict;
```

O ponto de entrada para o arquivo objeto compartilhado é considerado como sendo o mesmo que o nome da função SQL (aqui, *complexo_out*), a menos que especificado de outra maneira.

O exemplo a seguir continua aquele da seção anterior. A interface de programa de aplicação esconde a maior parte dos detalhes internos do PostgreSQL. Logo, o código C real para a função de saída de texto de valores do tipo *complexo* anterior é muito simples:

```
pg_function_info_v1(complexo_out);
Datum complexo_out(pg_function_args) {
    Complex *complexo = (Complex palloc(100);
    snprintf(result, 100, "(%g,%g)", complexo->x,
    complexo->y);
    pg_return_cstring(result);
}
```

As funções agregadas em PostgreSQL operam atualizando um **valor de estado** por meio de uma função de **transição de estado** que é chamada para cada valor de tupla no grupo de agregação. Por exemplo, o estado para o operador **avg** consiste na soma acumulada e na contagem de valores. À medida que cada tupla chega, a função de transição deve simplesmente somar seu valor à soma acumulada e incrementar o contador em um. Opcionalmente, uma função *final* pode ser chamada para calcular o valor de retorno com base na informação de estado. Por exemplo, a função final para **avg** simplesmente dividiria a soma acumulada pela contagem e a retornaria.

Assim, a definição de um novo operador é tão simples quanto a definição dessas duas funções. Para o exemplo de tipo *complexo*, se *soma_complexo* for uma função definida pelo usuário que apanha dois argumentos complexos e retorna sua soma, então o operador agregado **sum** pode ser estendido para números complexos usando a declaração simples:

```
create aggregate sum (
    sfunc = complexo_add,
    basetype = complexo,
    stype = complexo,
    initcond = '(0,0)'
);
```

Observe o uso da sobrecarga de função: PostgreSQL chamará a função de agregação *sum* apropriada, com base no

tipo real de seu argumento durante a invocação. O *basetype* é o tipo de argumento, e *stype* é o tipo de valor de estado. Nesse caso, uma função final é desnecessária, pois o valor de retorno é o próprio valor de estado (ou seja, a soma acumulada nos dois casos).

As funções definidas pelo usuário também podem ser invocadas usando a sintaxe de operador. Além do "açúcar sintático" simples para a invocação de funções, as declarações de operador também podem oferecer sugestões para o otimizador de consulta a fim de melhorar o desempenho. Essas sugestões podem incluir informações sobre comutatividade, restrição e estimativa de seletividade de junção, além de várias outras propriedades relacionadas a algoritmos de junção.

Extensões de índice

PostgreSQL aceita índices normais de árvore B e hash, além de índices de árvore R (para objetos espaciais bidimensionais) e índices GiST genéricos (que são exclusivos ao PostgreSQL e explicados na seção Tipos de índice"). Todos esses podem ser facilmente estendidos para acomodar novos tipos básicos.

A inclusão de extensões de índice para um tipo exige a definição de uma **classe de operador**, que encapsula o seguinte:

- **Estratégias de método de índice.** Estes são um conjunto de operadores que podem ser usados como qualificadores nas cláusulas **where**. O conjunto específico depende do tipo de índice. Por exemplo, índices de árvore B podem obter intervalos de objetos, de modo que o conjunto consiste em cinco operadores (<, <=, =, >= e >), todos aparecendo em uma cláusula **where** que envolve um índice de árvore B. Um índice de hash permite apenas o teste de igualdade, e um índice de árvore R permite uma série de relacionamentos espaciais (por exemplo, contido, à esquerda, e assim por diante).
- **Rotinas de suporte de método de índice.** O conjunto de operadores citado normalmente não é suficiente para a operação do índice. Por exemplo, um índice de hash exige que uma função calcule o valor de hash para cada objeto. Um índice de árvore R precisa ser capaz de calcular interseções e uniões e estimar o tamanho de objetos indexados.

Por exemplo, se as funções e operadores a seguir forem definidos para comparar a magnitude dos números do tipo *complexo* (ver seção "Tipos"), então podemos tornar tais objetos indexáveis com a seguinte declaração:

```
create operator class complexo_abs_ops
    default for type complexo using btree as
    operator 1  < (complexo, complexo),
    operator 2 <= (complexo, complexo),
    operator 3  = (complexo, complexo),
    operator 4 >= (complexo, complexo),
    operator 5  > (complexo, complexo),
    function 1 complexo_abs_cmp(complexo, complexo);
```

As instruções **operator** definem os métodos de estratégia e as instruções **function** definem os métodos de suporte.

Linguagens procedurais

Funções e procedimentos armazenados podem ser escritos em diversas linguagens de procedimento. Além do mais, PostgreSQL define uma interface de programa de aplicação para a ligação de qualquer linguagem de programação para essa finalidade. As linguagens de programação podem ser registradas por demanda e são **confiáveis** ou **não confiáveis**. As linguagens não confiáveis permitem acesso ilimitado ao SGBD e ao sistema de arquivos, e a escrita de funções armazenadas nelas exige privilégios de superusuário.

- **PL/pqSQL.** Esta é uma linguagem confiável que acrescenta capacidades de programação procedural (por exemplo, variáveis e fluxo de controle) à SQL. Ela é muito semelhante à PL/SQL do Oracle. Embora o código não possa ser transferido literalmente de uma para a outra, o transporte normalmente é simples.
- **PL/Tcl, PL/Perl e PL/Python.** Estas aproveitam o poder das linguagens Tcl, Perl e Python para escrever funções e procedimentos armazenados no servidor. As duas primeiras vêm em versões confiável e não confiável (PL/Tcl, PL/Perl e PL/TclU, PL/PerlU, respectivamente), enquanto PL/Python é não confiável no momento. Cada uma dessas possui vínculos que permitem o acesso ao sistema de banco de dados por meio de uma interface específica da linguagem.

Interface de programação do servidor

A interface de programação do servidor (SPI – Server Programming Interface) é uma API que permite que as funções C definidas pelo usuário (ver seção "Funções") executem comandos SQL arbitrariamente dentro de suas funções. Isso dá aos escritores de funções definidas pelo usuário a capacidade de implementar apenas as partes essenciais em C e aproveitar com facilidade o poder total do mecanismo do sistema de banco de dados relacional para fazer a maior parte do trabalho.

Gerenciamento de transações no PostgreSQL

O controle de concorrência do PostgreSQL implementa o controle de concorrência multiversão (MVCC – Multiver-

Nível de isolamento	*Leitura suja*	*Leitura não repetitiva*	*Fantasma*
Leitura não confirmada	Talvez	Talvez	Talvez
Leitura confirmada	Talvez	Talvez	Talvez
Leitura repetida	Talvez	Talvez	Talvez
Seriável	Não	Não	Não

Figura 26.5 Definição dos quatro níveis de isolamento SQL padrão.

sion Concurrency Control) e o bloqueio em duas fases. Qual dos dois protocolos é usado depende do tipo de instrução sendo executada. Para instruções DML,[1] é utilizado um esquema MVCC semelhante ao que apresentamos na seção "Ordenação por timestamp em múltipla versão" do Capítulo 16. O controle de concorrência para instruções DDL, por outro lado, é baseado no bloqueio padrão de duas fases.

Controle de concorrência do PostgreSQL

Como os detalhes do protocolo MVCC do PostgreSQL dependem do *nível de isolamento* solicitado pela aplicação, começamos com uma visão geral dos níveis de isolamento oferecidos pelo PostgreSQL. Depois, descrevemos as principais idéias por trás do esquema MVCC do PostgreSQL, seguidas pela discussão de sua implementação no MVCC do PostgreSQL e algumas implicações considerando o gerenciamento de armazenamento do PostgreSQL, o projeto de aplicações do usuário para PostgreSQL e o desempenho de um banco de dados PostgreSQL. Concluímos esta seção com uma visão geral do bloqueio para instruções DDL e uma discussão do controle de concorrência para índices.

Níveis de isolamento do PostgreSQL

O padrão SQL define três níveis de consistência fracos, além do nível de consistência seriável, em que a maior parte da discussão neste livro é baseada. O propósito de oferecer os níveis de consistência fracos é permitir um maior grau de concorrência para aplicações que não exigem as garantias fortes que a seriação oferece. Alguns exemplos de tais aplicações incluem transações de longa duração que coletam estatísticas sobre o banco de dados e cujos resultados não precisam ser exatos.

O padrão SQL define os diferentes níveis de isolamento em termos de três fenômenos que violam a seriação. Os três fenômenos são chamados de *leitura não repetitiva*, *leitura suja* e *leitura fantasma*, e são definidos da seguinte maneira:

- **Leitura não repetitiva**. Uma transação lê o mesmo objeto duas vezes durante a execução e encontra um valor diferente na segunda vez, embora a transação não tenha mudado o valor nesse meio tempo.
- **Leitura suja**. A transação lê valores escritos por outra transação que ainda não foi confirmada.
- **Leitura fantasma**. Uma transação reexecuta uma consulta retornando um conjunto de linhas que satisfazem uma condição de consulta e descobre que o conjunto de linhas que satisfazem a condição mudou como resultado de outra transação confirmada recentemente (uma explicação mais detalhada desse fenômeno pode ser encontrada na seção "O fenômeno fantasma" do Capítulo 16).

Deve ser óbvio que cada um desses fenômenos infringe o isolamento da transação e, portanto, infringe a seriação. A Figura 26.5 mostra a definição dos quatro níveis de isolamento SQL especificados no padrão SQL – leitura não confirmada, leitura confirmada, leitura repetitiva e seriável – em termos desses fenômenos. PostgreSQL admite dois dos quatro níveis de isolamento diferentes, leitura confirmada e seriável.

Controle de concorrência para comandos DML

A idéia básica por trás do MVCC é manter diferentes versões de uma linha que correspondam a diferentes instâncias da linha em diferentes pontos no tempo. O protocolo MVCC garante que cada transação verá apenas as versões dos dados que são consistentes com a visão da transação do banco de dados: cada transação vê um instantâneo dos dados, consistindo apenas nos dados que foram confirmados no momento em que a transação foi iniciada.[2] Esse instantâneo não é necessariamente igual ao estado atual dos dados.

1. Uma instrução DML é qualquer instrução que atualiza ou lê dados dentro de uma tabela, ou seja, **select**, **insert**, **update**, **fetch** e **copy**. Instruções DDL afetam tabelas inteiras; elas podem remover uma tabela ou alterar o esquema de uma tabela, por exemplo. Instruções DDL e algumas outras instruções específicas do PostgreSQL serão discutidas mais adiante nesta seção.

2. Além disso, as transações de consulta múltipla também vêem dados de consultas anteriores na mesma transação.

A motivação para usar o MVCC é que os leitores nunca bloqueiam os escritores e vice-versa. Os leitores acessam a versão mais recente de uma linha que faz parte do instantâneo da transação. Os escritores criam sua própria cópia separada da linha a ser atualizada. A próxima seção mostra que o único conflito que faz com que uma transação seja bloqueada surge se dois escritores tentarem atualizar a mesma linha. Ao contrário, sob a técnica de bloqueio em duas fases, leitores e escritores poderiam ser bloqueados, pois há somente uma versão de cada objeto de dados, e as operações de leitura e escrita precisam obter um bloqueio antes de acessar quaisquer dados.

O MVCC do PostgreSQL é muito semelhante em espírito ao esquema de ordenação de timestamp multiversão descrito na seção "Ordenação por timestamp em múltipla versão" do Capítulo 16. PostgreSQL nunca emprega os bloqueios explicitamente para comandos DML, e, portanto, nenhuma interação com um gerenciador de bloqueio será necessária. Isso difere do esquema MVCC empregado pelo Oracle, o único sistema de banco de dados comercial que utiliza MVCC em vez do bloqueio em duas fases. O esquema MVCC do Oracle é basicamente o protocolo de bloqueio em duas fases multiversão descrito na seção "Bloqueio de duas fases em múltipla versão" do Capítulo 16.

Implementação PostgreSQL do MVCC

No núcleo do MVCC do PostgreSQL está a noção de *visibilidade de tupla*. Uma tupla PostgreSQL refere-se a uma versão de uma linha. A visibilidade de tupla define qual das potencialmente muitas versões de uma linha em uma tabela é válida dentro do contexto de determinada instrução ou transação.

Uma tupla é visível para uma transação T_A se estas duas condições forem verdadeiras:

1. A tupla foi criada por uma transação T_B que começou sua execução e foi confirmada antes que a transação T_A começasse a executar.
2. As atualizações feitas à tupla (se houver) foram executadas por uma transação T_C que
   - É abortada *ou*
   - Começou a executar após a transação T_A *ou*
   - Estava em processo no início de T_A.

O objetivo dessas condições é assegurar que cada transação acesse apenas dados que foram confirmados no momento em que a transação iniciou sua execução. PostgreSQL mantém as seguintes estruturas de dados para verificar essas condições de forma eficiente:

- Uma *ID de transação*, que ao mesmo tempo serve como timestamp, é atribuída a cada transação no momento da partida da transação. PostgreSQL utiliza um contador lógico (conforme descrito na seção "Timestamp" do Capítulo 16) para atribuir IDs de transação.
- Um arquivo de log chamado *pg-clog* contém o status atual de cada transação. O status pode ser em progresso, confirmado ou abortado.
- Cada tupla em uma tabela possui um cabeçalho com três campos: *xmin*, que contém a ID da transação que criou a tupla e que, portanto, também é chamada de *ID de transação de criação*; *xmax*, que contém a ID da transação substituindo/excluindo (ou *nulo*, se não for excluído/replicado) e que também é referenciada como *ID de transação de expiração*; e um link para a frente, para novas versões da mesma linha lógica, se houver alguma.
- Uma estrutura de dados *SnapshotData* é criada na hora do início da transação ou na hora do início da consulta, dependendo do nível de isolamento (descrito com mais detalhes a seguir). A estrutura de dados *SnapshotData* contém, entre outras coisas, uma lista de todas as transações que estão ativas no momento em que o instantâneo é apanhado.

A Figura 26.6 ilustra essas estruturas de dados por meio de um exemplo simples, envolvendo um banco de dados com apenas uma tabela, a tabela *agência* da Figura 2.3. A tabela *agência* contém três atributos, o nome da agência, a cidade onde a agência está localizada e os ativos da agência. A Figura 26.6 mostra um fragmento da tabela de agências contendo apenas as versões de linha correspondentes à agência "Brighton". Os cabeçalhos de tupla indicam que a linha foi criada originalmente pela transação 100 e atualizada mais tarde pelas transações 102 e 106. A Figura 26.6 também mostra um fragmento do arquivo *pg_log* correspondente. Com base no arquivo *pg_log*, as transações 100 e 102 são confirmadas, enquanto as transações 104 e 106 estão em progresso.

Dadas essas estruturas de dados, as duas condições que precisam ser satisfeitas para uma tupla ser visível podem ser reescritas da seguinte forma:

1. A ID de transação de criação no cabeçalho de tupla
   - É uma transação confirmada de acordo com o arquivo *pg_log* e
   - É menor que o contador de transação armazenado no início da consulta em *SnapshotData* e
   - Não estava em processo no início da consulta, de acordo com *SnapshotData*.
2. A ID de transação de expiração
   - Está em branco ou é abortada *ou*
   - É maior do que o contador de transação armazenado no início da consulta em *SnapshotData ou*

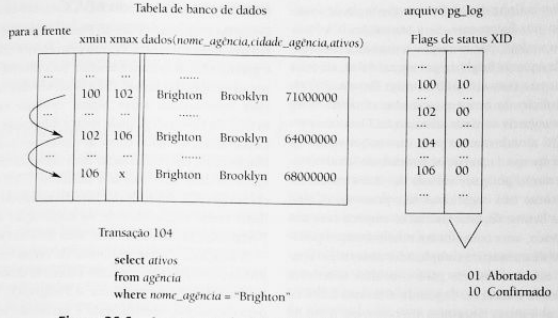


Figura 26.6 As estrutura de dados do PostgreSQL usadas para MVCC.

- Estava em processo no início da consulta de acordo com *SnapshotData*.

Por exemplo, a única versão da linha correspondente à agência "Brighton" que é visível à transação 104 na Figura 26.6, é a segunda versão na tabela, criada pela transação 102. A primeira versão, criada pela transação 100, não é visível, pois infringe a condição 2: a ID de transação de expiração dessa tupla é 102, que corresponde a uma transação que não é abortada e que tem uma ID de transação menor que a transação 104. A terceira versão dessa agência "Brighton" não é visível, pois foi criada pela transação 106, que possui uma ID de transação maior que a transação 104, implicando que essa versão não foi confirmada no momento em que a transação 104 começou a executar. Além do mais, a transação 106 ainda está em progresso, o que infringe outra das condições. A segunda versão da linha atende a todas as condições para visibilidade de tupla.

Os detalhes de como o MVCC do PostgreSQL interage com a execução das instruções SQL dependem se a instrução é uma instrução **insert**, **select**, **update** ou **delete**. O caso mais simples é a instrução **insert**. Diferente do caso do bloqueio de duas fases, para uma instrução **insert**, basicamente não existe interação com o protocolo de controle de concorrência durante a execução da instrução: uma instrução **insert** simplesmente cria uma nova tupla com base nos dados da instrução, inicializa o cabeçalho da tupla (a ID de criação) e insere a nova tupla na tabela.

Quando o sistema executa uma instrução **select**, **update** ou **delete**, a interação com o protocolo MVCC depende do nível de isolamento especificado pela aplicação. Se o nível de isolamento for confirmado para leitura, o processamento de uma nova instrução começa com a criação de uma nova estrutura de dados *SnapshotData* (independente de se a instrução inicia uma nova transação ou faz parte de uma transação existente). Em seguida, o sistema identifica *tuplas de destino*; ou seja, as tuplas que são visíveis com relação a *SnapshotData* e que combinam com os critérios de busca da instrução. No caso de uma instrução **select**, o conjunto de tuplas de destino compõe o resultado da consulta.

No caso de uma instrução **update** ou **delete**, um passo extra é necessário após a identificação das tuplas de destino, antes que a operação real de atualização ou exclusão possa ocorrer. O motivo é que a visibilidade de uma tupla garante apenas que a tupla tenha sido criada por uma transação que foi confirmada antes que a instrução **update/delete** em questão seja iniciada. Porém, é possível que, desde o início da consulta, essa tupla tenha sido atualizada ou excluída por outra transação executando simultaneamente. Isso pode ser detectado examinando-se a ID de transação de expiração da tupla. Se a ID de transação de expiração corresponde a uma transação que ainda está em andamento, é preciso esperar primeiro pelo término dessa transação. Se a transação for abortada, a instrução **update** ou **delete** pode prosseguir e realizar a operação update/delete real. Se a transação for confirmada, o critério de busca da instrução **update/delete** precisa ser avaliado novamente, e somente se a tupla ainda atender a esses critérios, o update/delete pode ser realizado. Realizar a operação update/delete inclui a criação de uma nova tupla (com o cabeçalho correspondente contendo a ID de criação) e também a atualização das informações de cabeçalho da tupla antiga (ou seja, a ID de transação de expiração).

Voltando ao exemplo da Figura 26.6, a transação 104, que consiste em uma instrução **select** apenas, identifica a

segunda versão da linha de Brighton como tupla de destino e a retorna imediatamente. Se a transação 104 fosse uma instrução update, por exemplo, tentando incrementar os ativos da agência Brighton por algum valor, ela teria de esperar até que a transação 106 termine. Depois, ela reavaliaria a condição de busca e, somente se ainda fosse atendida, prosseguiria com sua atualização. O uso do protocolo descrito anteriormente para instruções **update** e **delete** oferece apenas o nível de isolamento de leitura confirmada. A seriação pode ser violada de várias maneiras. Primeiro, leituras não repetitivas são possíveis. Como cada consulta dentro de uma transação começa com um novo instantâneo, uma consulta em uma transação poderia ver o efeito de transações completadas nesse meio tempo, que não estavam visíveis para consultas anteriores dentro da mesma transação. Seguindo a mesma linha de pensamento, as leituras fantasmas são possíveis. Além do mais, uma consulta **update** pode ver os efeitos das atualizações concorrentes por outras consultas feitas à mesma linha, mas não ver o efeito das consultas concorrentes sobre outras linhas no banco de dados.

Para oferecer o nível de isolamento seriável do PostgreSQL, o MVCC do PostgreSQL elimina violações de seriação de duas maneiras: primeiro, quando ele determina a visibilidade da tupla, todas as consultas dentro de uma transação utilizam um instantâneo como início da transação, em vez do início da consulta individual. Desse modo, consultas sucessivas dentro de uma transação sempre vêem os mesmos dados.

Em segundo lugar, o modo como as atualizações e exclusões são processadas é diferente no modo seriável em comparação com o modo de leitura confirmada. Como no modo de leitura confirmada, as transações esperam, depois de identificar uma linha de destino visível que atenda a condição de busca e seja atualmente atualizada ou excluída por outra transação concorrente. Se a transação concorrente que executa o update/delete abortar, a transação esperando pode prosseguir com sua própria atualização. Porém, se a transação concorrente for confirmada, não haverá como o PostgreSQL garantir a seriação para a transação esperando. Portanto, a transação esperando é revertida e retorna com a mensagem de erro "Can't serialize access due to concurrent update" (impossível seriar o acesso devido a atualização concorrente).

Fica a critério da aplicação tratar corretamente de uma mensagem de erro como essa, abortando a transação atual e reiniciando a transação inteira desde o início. Observe que os rollbacks decorrentes de questões de seriação só são possíveis para instruções update. As transações somente leitura ainda não poderão entrar em conflito com quaisquer outras transações.

Implicações do uso do MVCC

O uso do esquema MVCC do PostgreSQL tem implicações em três áreas diferentes: (1) um peso extra é colocado sobre o gerenciador de armazenamento, pois precisa manter diferentes versões das tuplas; (2) o desenvolvimento de aplicações concorrentes exige algum cuidado extra, pois o MVCC do PostgreSQL pode levar a diferenças sutis, porém importantes, no modo como as transações concorrentes se comportam, em comparação com sistemas em que o bloqueio de duas fases padrão é utilizado; (3) o desempenho do PostgreSQL depende das características da carga de trabalho sendo executada nele. As implicações do MVCC do PostgreSQL são descritas com mais detalhes a seguir.

A criação e o armazenamento de várias versões de cada linha podem levar ao consumo excessivo de armazenamento. Para aliviar esse problema, o PostgreSQL libera espaço periodicamente, identificando e excluindo versões das linhas que não são mais necessárias. Essa funcionalidade é implementada na forma do comando **vacuum**. O comando **vacuum** é executado como um daemon em segundo plano, mas também pode ser chamado diretamente pelo usuário.

O comando **vacuum** oferece diferentes modos de operações: **vacuum** simplesmente reivindica o espaço ocupado pelas linhas não mais usadas, e torna esse espaço disponível para reutilização. Essa forma do comando pode operar em paralelo com a leitura e escrita normal da tabela. **Vacuum full** realiza um processamento mais extenso, incluindo a movimentação de tuplas por blocos para tentar compactar a tabela para o número mínimo de blocos de disco. Essa forma é muito mais lenta e exige um bloqueio exclusivo em cada tabela enquanto está sendo processada. Quando chamado com o parâmetro opcional **analyze**, **vacuum** reunirá estatísticas sobre o conteúdo das tabelas que está "aspirando". Os resultados são utilizados para atualizar a tabela do sistema *pg_statistic*, permitindo que o planejador de consulta do PostgreSQL faça melhores escolhas no planejamento das consultas.

Devido ao uso do controle de concorrência de versão múltipla no PostgreSQL, o transporte de aplicações de outros ambientes para o PostgreSQL pode exigir algum cuidado extra para garantir a consistência dos dados. Como exemplo, considere uma transação T_A executando uma instrução **select**. Como os leitores em PostgreSQL não bloqueiam dados, os dados lidos e selecionados por T_A podem ser reescritos por outra transação concorrente T_B, enquanto T_A ainda está sendo executada. Como resultado, alguns dos dados que T_A retorna podem não estar mais atualizados no momento do término de T_A. T_A poderia retornar linhas que, nesse meio tempo, foram alteradas ou excluídas por outras transações. Para garantir a validade atual de uma linha e protegê-la contra atualizações concorrentes, uma aplicação precisa usar **select for update** ou adquirir explicitamente um bloqueio com o comando **lock table** apropriado.

A técnica do PostgreSQL para o controle de concorrência oferece desempenho ideal para cargas de trabalho contendo muito mais leituras do que atualizações, pois nesse caso as chances de que duas atualizações entrem em conflito, resultando no rollback de uma transação, são muito baixas.

Controle de concorrência da DDL

Os mecanismos de MVCC descritos na seção anterior não protegem transações contra operações que afetam tabelas inteiras, por exemplo, transações que descartam uma tabela ou mudam o esquema de uma tabela. Para essa finalidade, PostgreSQL oferece bloqueios explícitos que os comandos DDL são forçados a adquirir antes de iniciar sua execução. Esses bloqueios sempre são baseados em tabela (e não em linha) e são adquiridos e liberados de acordo com o protocolo estrito de bloqueio em duas fases.

A Figura 26.7 lista todos os tipos de bloqueios oferecidos pelo PostgreSQL, os comandos que os utilizam e a compatibilidade com outros modos de bloqueio. Os nomes dos tipos de bloqueio normalmente são históricos e não necessariamente refletem o uso do bloqueio. Por exemplo, todos os bloqueios são bloqueios em nível de tabela, embora alguns contenham a palavra "row" (linha) no nome. Os comandos da DML adquirem apenas bloqueios dos tipos 1, 2 ou 3. Esses três tipos de bloqueio são compatíveis entre si, pois o MVCC cuida de proteger essas operações umas contra as outras. Os comandos da DDL só adquirem esses bloqueios para proteção contra comandos da DDL.

Embora sua finalidade principal seja oferecer controle de concorrência interno do PostgreSQL para comandos da DDL, todos os bloqueios na Figura 26.7 também podem ser adquiridos explicitamente pelas aplicações PostgreSQL por meio do comando **lock table**.

Os bloqueios são registrados em uma tabela de bloqueios que é implementada como uma tabela de hash da memória compartilhada tendo como chave o tipo e a ID do objeto sendo bloqueado. Se uma transação quiser adquirir um bloqueio sobre um objeto mantido por outra transação no modo não compatível, ela precisa esperar até que o bloqueio seja liberado. Esperas de bloqueio são implementadas por meio de semáforos. Cada transação tem um semáforo asso-

Adquirido por	*Conflita com*	*Nome do bloqueio*
ACCESS SHARE	ACCESS EXCLUSIVE	**select** query
ROW SHARE	EXCLUSIVE ACCESS EXCLUSIVE	**select for update** query
ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**update** **delete** **insert** queries
SHARE UPDATE EXCLUSIVE	SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**vacuum**
SHARE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE ACCESS EXCLUSIVE	**create index**
SHARE ROW EXCLUSIVE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE ACCESS EXCLUSIVE	—
EXCLUSIVE	ROW SHARE ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE ACCESS EXCLUSIVE	—
ACCESS EXCLUSIVE	CONFLICTS WITH LOCKS OF ALL MODES	**drop table** **alter table** **vaccum full**

Figura 26.7 Modos de bloqueio em nível de tabela.

ciado a ele. Ao esperar por um bloqueio, uma transação realmente espera pelo semáforo associado à transação que mantém o bloqueio. Quando o mantenedor do bloqueio liberá-lo, ele sinalizará as transações esperando por meio do semáforo. Implementando esperas de bloqueio com base em cada mantenedor de bloqueio, no lugar do objeto do bloqueio, o PostgreSQL exige no máximo um semáforo por transação concorrente, em vez de um semáforo por objeto bloqueável.

A detecção de impasse no PostgreSQL é baseada em timeouts. Como padrão, a detecção de impasse é disparada se uma transação estiver esperando por um bloqueio por mais de um segundo. O algoritmo de detecção de impasse constrói um gráfico de espera baseado na informação da tabela de bloqueio e pesquisa esse gráfico em busca de dependências circulares. Se encontrar alguma, significando que foi detectado um impasse, a transação que disparou a detecção do impasse é abortada e retorna um erro ao usuário. Se nenhum ciclo for detectado, a transação continua esperando pelo bloqueio. Ao contrário de alguns sistemas comerciais, o PostgreSQL não ajusta dinamicamente o parâmetro de timeout do bloqueio, mas permite que o administrador o ajuste manualmente. O ideal é que esse parâmetro deva ser escolhido na ordem do tempo de vida de uma transação, a fim de otimizar a escolha entre o tempo gasto para detectar um impasse e o trabalho desperdiçado para executar o algoritmo de detecção de impasse sempre que não houver impasse.

Bloqueio e índices

O bloqueio realizado para acessos a índice depende do tipo de índice. Para GiST e árvores R, bloqueios simples no nível de índice são usados e mantidos por toda a duração do comando. Os acessos de índice de hash levam em conta a concorrência mais alta usando bloqueios em nível de página que são liberados após a página ser processada. Porém, o bloqueio no nível de página para índices de hash não é livre de impasse. O tipo de índice recomendado para aplicações com um alto grau de concorrência são os índices de árvore B, pois oferece bloqueio detalhado sem condições de impasse: os acessos às árvores B são protegidos por bloqueios de nível de página compartilhados/exclusivos a curto prazo. Os bloqueios são liberados imediatamente após cada tupla de índice ser apanhada ou inserida.

Recuperação

Historicamente, PostgreSQL não usava o logging de escrita antecipada para recuperação, e por isso não era capaz de garantir a consistência no caso de falha. Uma falha potencialmente poderia resultar em estruturas de dados inconsistentes ou, pior ainda, conteúdo de tabela totalmente adulterado, devido a páginas de dados parcialmente escritas. Como resultado, a partir da versão 7.1, PostgreSQL emprega a recuperação padrão baseada em log de escrita antecipada com uma fase de redo e uma fase de undo, semelhante ao ARIES. A seção "Buffering de registro de log" **do Capítulo 17** oferece uma introdução de redo e undo e o conceito de logging de escrita antecipada, e a seção "ARIES" **do Capítulo 17** oferece uma visão geral do ARIES. A principal diferença na implementação PostgreSQL da recuperação é que o MVCC do PostgreSQL permite algumas simplificações em comparação com a recuperação padrão baseada em log.

Em primeiro lugar, sob o PostgreSQL, a recuperação não precisa desfazer os efeitos das transações abortadas: uma transação abortando faz uma entrada no arquivo pg-clog, registrando o fato de estar abortando. Conseqüentemente, todas as versões das linhas que ele deixa para trás nunca serão visíveis a quaisquer outras transações. O único caso em que essa técnica potencialmente poderia levar a problemas é quando uma transação aborta devido a uma falha do processo PostgreSQL correspondente e o processo PostgreSQL não tem uma chance de criar a entrada *pc_clog* antes da falha. O PostgreSQL trata disso da seguinte maneira: sempre que uma transação verifica o status de outra transação no arquivo pg-clog e descobre que o status é "em andamento", ela verifica se a transação está realmente executando em qualquer um dos processos do PostgreSQL. Se nenhum processo do PostgreSQL estiver atualmente executando a transação, será deduzido que o processo correspondente falhou, e a entrada *pg-clog* da transação é atualizada para "abortada".

Em segundo lugar, a recuperação baseada no log de escrita antecipada do PostgreSQL é simplificada pelo fato de que o MVCC do PostgreSQL registra alguma informação exigida pelo registro do WAL. Mais precisamente, não há necessidade de registrar as transações de início, confirmação e aborto, pois o MVCC registra o status de cada transação no *pc_clog*.

Armazenamento e indexação

Acompanhando o restante do PostgreSQL, a filosofia de projeto do layout e armazenamento de dados visa os objetivos gêmeos de (1) uma implementação simples e clara e (2) facilidade de administração. Os bancos de dados gerenciados por um servidor PostgreSQL são particionados pelo administrador em *clusters de banco de dados*, em que todos os dados e metadados associados a um cluster são armazenados no mesmo diretório do sistema de arquivos. Ao contrário dos sistemas comerciais, o PostgreSQL não admite "tablespaces",[3] que dariam ao administrador de banco de dados o controle preciso do local de armazenamento de cada objeto físico individual. Além disso, PostgreSQL admite

3. O desenvolvimento ativo de tablespaces é contínuo e fará parte de uma versão futura.

apenas "sistemas de arquivo preparados", evitando o uso de partições de disco brutas.

A simplicidade no projeto do sistema de armazenamento do PostgreSQL potencialmente leva a algumas limitações de desempenho. A falta de suporte para tablespaces limita as possibilidades de usar de forma eficiente os recursos de armazenamento disponíveis, em particular, vários discos operando em paralelo. Além do mais, o tamanho de bloco de todos os objetos do banco de dados é fixo (o padrão é de 8 kilobytes e pode ser alterado pela recompilação do código), que pode oferecer desempenho inferior ao lidar com armazenamento que trata de blocos de dados maiores. O uso de sistemas de arquivos preparados resulta no buffer duplo, em que um bloco é primeiro lido do disco para o cache do sistema de arquivos (no espaço do kernel) antes de ser copiado para o pool de buffer do PostgreSQL.

Por outro lado, as empresas modernas cada vez mais utilizam sistemas de armazenamento, como o armazenamento conectado à rede e as redes de área de armazenamento, no lugar de discos conectados a servidores. A filosofia aqui é que o armazenamento é um serviço facilmente administrado e ajustado para desempenho separadamente. O uso de RAID para conseguir paralelismo e armazenamento redundante é explicado na seção "RAID" do Capítulo 11.

Assim, o sentimento de muitos desenvolvedores PostgreSQL é que, para uma grande maioria das aplicações, e na realidade a audiência do PostgreSQL, as limitações de desempenho são mínimas e justificadas pela facilidade de administração e gerenciamento, além da simplicidade de implementação.

Tabelas

A principal unidade de armazenamento no PostgreSQL é uma tabela. Em PostgreSQL, a tabelas são armazenadas em "arquivos de heap". Esses arquivos usam um formato padrão de "página em slots", descrito na seção "Registros de tamanho variável" do Capítulo 11. O formato do PostgreSQL aparece na Figura 26.8. Em cada página, um cabeçalho é seguido por um array de "ponteiros de linha". Um ponteiro de linha mantém o deslocamento (relativo ao início da página) e o tamanho de uma tupla específica na página. As tuplas reais são armazenadas na ordem reversa dos ponteiros de linha a partir do final da página.

Um registro em um arquivo de heap é identificado por sua **ID de tupla** (TID). A TID consiste em uma ID de bloco de 4 bytes que especificam a página no arquivo contendo a tupla e uma ID de slot de 2 bytes. A ID de slot é um índice para o array de ponteiro de linha que, por sua vez, é usado para acessar a tupla.

Embora essa infra-estrutura permita que tuplas em uma página sejam excluídas ou atualizadas, sob a técnica MVCC do PostgreSQL, nenhuma operação realmente ocorre fisicamente durante o processamento normal. Quando uma página é *aspirada*, porém, as tuplas expiradas são fisicamente excluídas, fazendo com que sejam formados buracos em uma página. A indireção do acesso às tuplas por meio do array de ponteiro de linha permite a reutilização desses furos.

O tamanho de uma tupla normalmente é limitado pelo tamanho de uma página de dados. Isso dificulta armazenar tuplas muito longas. Quando o PostgreSQL encontra uma tupla grande, ele tenta **tostar** colunas grandes individuais. Os dados em uma coluna tostada são substituídos por um ponteiro que localiza uma versão compactada dos dados que são armazenados fora da página.

Índices

PostgreSQL admite vários índices diferentes, incluindo aqueles que são baseados nos métodos de acesso extensí-

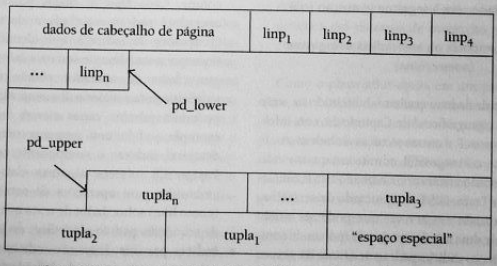


Figura 26.8 Formato de página em slots para tabelas PostgreSQL.

veis ao usuário. Embora um método de acesso possa usar um formato de página diferente, todos os índices disponíveis no PostgreSQL usam o formato de página em slots, já descrito na seção anterior.

Tipos de índice

PostgreSQL aceita os seguintes tipos de índices:

- **Árvore B**: o tipo de índice-padrão é um método de árvore B que é uma implementação das árvores B de alta concorrência de Lehman-Yao (isso é explicado com detalhes na seção "Concorrência em estruturas de índice" do Capítulo 16). Esses índices são úteis para consultas de igualdade e intervalo sobre dados classificáveis e também para certas operações de combinação de padrão, como a expressão **like**.
- **Hash**: os índices de hash do PostgreSQL são uma implementação do hashing linear (para obter mais informações sobre índices de hash, consulte a seção "Índices sobre chaves múltiplas" do Capítulo 12). Esses índices são úteis apenas para operações de igualdade simples. Um índice de hash pode ser criado com a seguinte instrução DDL:

create index *nomehandx* **on** *nometabela* **using hash** (*nomecoluna*)

Os índices de hash do PostgreSQL não têm apresentado desempenho de pesquisa melhor do que o das árvores B, mas possuem tamanho e custos de manutenção muito maiores. Assim, quase sempre é preferível usar índices de árvore B em vez de índices de hash.
- **Árvore R**: para operações como *sobreposição* nos tipos de dados espaciais internos (box, circle, point e assim por diante), PostgreSQL oferece índices de árvore R. Estes implementam o algoritmo de divisão quadrática (seção "Árvores R" do Capítulo 24). Um índice de árvore R pode ser criado com a seguinte instrução DDL:

create index *nomerndx* **on** *nometabela* **using rtree** (*nomecoluna*)

O gerenciamento de dados espaciais foi discutido na seção "Dados espaciais e geográficos" do Capítulo 24, com informações sobre árvores R e outras técnicas de indexação.
- **GiST**: finalmente, o PostgreSQL admite um quarto índice extensível baseado em árvore, chamado GiST, ou Generalized Search Trees. GiST é um método de acesso balanceado, estruturado em árvore, que pode ser usado para implementar uma família inteira de índices diferentes – por exemplo, os índices padrão de árvore B e árvore R podem ser implementados usando GiST. Índices GiST facilitam para um especialista em domínio, bem versado em determinado tipo de dados (como dados de imagem), desenvolver índices de melhoria de desempenho sem ter de tratar dos detalhes internos do sistema de banco de dados. Exemplos de alguns índices embutidos usando GiST incluem a indexação para cubos multidimensionais e indexação de texto completa para consultas de recuperação de informações. Consulte as notas bibliográficas para obter referências a mais informações sobre o índice GiST.

Outras variações de índice

Para alguns dos tipos de índice descritos aqui, o PostgreSQL admite variações mais complexas, como:

- **Índices de múltiplas colunas**. Estes são úteis para conjuntos de predicados por várias colunas de uma tabela. Os índices de múltiplas colunas são aceitos apenas para índices de árvore B e GiST.
- **Índices exclusivos**. As restrições exclusivas e de chave primária podem ser impostas pelo uso de índices exclusivos no PostgreSQL. Somente índices de árvore B podem ser definidos como sendo exclusivos.
- **Índices sobre expressões**. Em PostgreSQL, é possível criar índices sobre quaisquer expressões escalares de colunas, e não apenas colunas específicas de uma tabela. Isso é especialmente útil quando as expressões em questão são "dispendiosas" – digamos, envolvendo cálculos complicados definidos pelo usuário. Um exemplo é o suporte a comparações sem diferenciar maiúsculas e minúsculas usando o predicado *lower*(*coluna*) = 'valor' nas consultas. Uma desvantagem é que os custos de manutenção de índices sobre expressões são muito altos.
- **Classes de operadores**. As funções de comparação específicas usadas para criar, manter e usar um índice sobre uma coluna estão ligadas ao tipo de dados dessa coluna. Cada tipo de dados possui uma "classe de operador" padrão associada a ele (descrita na seção "Extensões de índice"), que identifica os operadores reais que normalmente seriam usados para ele. Embora essa classe de operador padrão normalmente seja suficiente para a maioria dos usos, alguns tipos de dados poderiam possuir várias classes "significativas". Por exemplo, ao lidar com números complexos, poderia ser desejável indexar o componente real ou imaginário. PostgreSQL oferece algumas classes de operadores embutidas para operações de combinação de padrão (como **like**) sobre dados de texto que não usam as regras de sequência padrão específicas do local.
- **Índices parciais**. Estes são índices embutidos por um subconjunto de uma tupla definida por um predicado. O

índice contém apenas entradas para tuplas que satisfazem o predicado. Os índices parciais são adequados para casos em que uma coluna poderia conter uma grande quantidade de ocorrências de um número muito pequeno de valores. Nesses casos, sem um índice parcial, uma consulta de "agulha no palheiro" que procura um valor incomum acabaria varrendo o índice inteiro. Um índice parcial que exclui os valores comuns é pequeno e incorre em menos E/S. Os índices parciais são menos dispendiosos de manter, pois uma grande fração das inserções não participa do índice.

Processamento e otimização de consulta

Quando o PostgreSQL recebe uma consulta, ela primeiro é passada para uma representação interna, que passa por uma série de transformações, resultando em um plano de consulta que é usado pelo **executor** para processar a consulta.

Reescrita de consulta

O primeiro estágio da transformação de uma consulta é **rewrite**, responsável pelo sistema de **rules** do PostgreSQL. Conforme explicamos na seção "Regras e outros recursos de banco de dados ativo", no PostgreSQL, os usuários podem criar **rules** que são disparadas em diferentes eventos, como instruções **update**, **delete**, **insert** e **select**. Uma view é implementada pelo sistema convertendo uma definição de view para uma regra **select**. Quando uma consulta envolvendo uma instrução **select** sobre a view é recebida, a regra **select** para a view é disparada, e a consulta é reescrita usando a definição da view.

Uma regra é registrada no sistema usando o comando **create rule**, no ponto em que a informação sobre a regra é armazenada no catálogo. Esse catálogo é então usado durante a reescrita da consulta para desvendar todas as regras candidatas para determinada consulta.

A fase de reescrita primeiro lida com todas as instruções **update**, **delete** e **insert** disparando todas as regras apropriadas. Observe que essas instruções poderiam ser complicadas e conter cláusulas **select**. Mais tarde, todas as regras restantes envolvendo apenas instruções **select** são disparadas. Como o disparo de uma regra pode fazer com que a consulta seja reescrita para uma forma que pode exigir o disparo de outra regra, as regras são repetidamente verificadas em cada forma da consulta reescrita até que um ponto fixo seja alcançado e nenhuma outra regra precise ser disparada.

Não existem regras padrão no PostgreSQL – apenas aquelas definidas explicitamente pelos usuários e implicitamente pela definição de views.

Planejamento e otimização de consulta

Quando a consulta tiver sido reescrita, ela estará sujeita à fase de planejamento e otimização. Aqui, cada bloco de consulta é tratado em isolado e um plano é gerado para ele. Esse planejamento começa de baixo para cima, a partir da subconsulta mais interna da consulta reescrita, prosseguindo para o bloco de consulta mais externo.

O otimizador no PostgreSQL é, em sua maior parte, baseado em custo. O ideal é gerar um plano de acesso cujo custo estimado é mínimo. O modelo de custo inclui como parâmetros o custo de E/S de uma busca de página não sequencial e também os custos de CPU do processamento de tuplas de heap, tuplas de índice e predicados simples.

O processo real de otimização é baseado em uma das duas formas a seguir:

- **Planejador padrão.** Essa é a técnica de otimização tradicional usada no System R, um sistema relacional pioneiro, desenvolvido pela IBM na década de 1970. Esse é um algoritmo de programação dinâmico em que, para cada bloco, todas as possibilidades de junção bidirecionais são enumeradas, planejadas e estimadas pelo custo de acesso. Depois, todas as possibilidades de junção tridirecionais são enumeradas e estimadas, usando as melhores estimativas de junção bidirecionais. Esse processo continua até que um plano "bom" para o bloco de consulta seja produzido. Outros detalhes sobre essa técnica estão no Capítulo 14.
- **Otimizador de consulta genético.** Quando o número de tabelas em um bloco de consulta for muito grande, o algoritmo de programação dinâmica do System R se torna muito dispendioso. Ao contrário de outros sistemas comerciais que usam como padrão algoritmos gulosos, técnicas greedy ou baseadas em regras, PostgreSQL utiliza uma técnica mais radical: um algoritmo genético que foi desenvolvido inicialmente para solucionar problemas de vendedores em viagem. Existe evidência anedótica do uso bem-sucedido da otimização de consulta genética em sistemas de produção para consultas com cerca de 45 tabelas.

Como o planejador opera em um padrão de baixo para cima, ele é capaz de realizar certas transformações sobre o plano de consulta enquanto está sendo criado. Um exemplo é a transformação comum de subconsulta para junção, que está presente em muitos sistemas comerciais (normalmente implementados na fase de reescrita). Quando o PostgreSQL encontra uma subconsulta não correlacionada (como aquela causada por uma consulta sobre uma view), geralmente é possível "puxar" a subconsulta planejada e mesclá-la ao bloco de consulta de nível superior. Porém, transformações que empurram a eliminação distinta para

os blocos de consulta de nível inferior geralmente não são possíveis em PostgreSQL.

A fase de otimização de consulta resulta em um plano de consulta que é uma árvore de operadores relacionais. Cada operador representa uma operação específica sobre um ou mais conjuntos de tuplas. Os operadores podem ser unários (por exemplo, classificação, agregação), binários (por exemplo, junção de loop aninhado) ou *n*-ários (por exemplo, união de conjunto).

Crucial para o modelo de custo é uma estimativa precisa do número total de tuplas que serão processadas em cada operador no plano. Esta é deduzida pelo otimizador com base nas estatísticas que são mantidas em cada relação no sistema. Estas indicam o número total de tuplas para cada relação e informações específicas sobre cada coluna de uma relação, como a cardinalidade da coluna, uma lista dos valores mais comuns na tabela e o número de ocorrências, com um histograma que divide os valores da coluna em grupos de mesma população. Além disso, PostgreSQL também mantém uma correlação estatística entre as ordens de linha física e lógica dos valores de uma coluna – isso indica o custo de uma varredura de índice para recuperar tuplas que passam predicados sobre a coluna. O DBA precisa garantir que essas estatísticas sejam atualizadas, executando o comando **analyze** periodicamente.

Executor de consulta

O módulo executor é responsável por processar um plano de consulta produzido pelo otimizador. O executor segue o modelo do **iterador** com um conjunto de quatro funções implementadas para cada operador (`open`, `next`, `rescan` e `close`). Os iteradores também são discutidos como parte da canalização controlada por demanda na seção "Implementação da canalização" do Capítulo 13. Iteradores do PostgreSQL possuem uma função extra, `rescan`, que é usada para reiniciar um subplano (digamos, para um loop interno de uma junção) com parâmetros como intervalos de chave de índice.

Alguns dos operadores importantes do executor podem ser categorizados da seguinte maneira:

1. **Métodos de acesso.** Os métodos de acesso reais que são usados para obter dados de objetos em disco no PostgreSQL são varreduras seqüenciais a partir de varreduras de heap e índice.
   - **Varreduras seqüenciais.** As tuplas de uma relação são varridas seqüencialmente do primeiro ao último bloco do arquivo. Cada tupla é retornada a quem chamou somente se for "visível" de acordo com as regras de isolamento de transação na seção "Implementação PostgreSQL do MVCC".
   - **Varreduras de índice.** Dado um intervalo de índice (ou uma chave específica no caso de um índice de heap), esse método de acesso retorna um conjunto de tuplas que combinam a partir do arquivo de heap associado. Não há provisão para separar o acesso às IDs de tupla a partir do índice e apanhar as tuplas reais. Isso impede a classificação das TIDs e garante que o arquivo de heap seja acessado de maneira seqüencial, reduzindo o número de operações de busca de página.
2. **Métodos de junção.** O PostgreSQL admite três métodos de junção: junções merge classificadas, junções de loop aninhado (incluindo variantes de loop aninhado indexado para a junção interna) e uma junção de hash híbrida.
3. **Sort.** A classificação externa é implementada no PostgreSQL por algoritmos explicados na seção "Classificação" do Capítulo 13. A entrada é dividida em execuções classificadas que são então mescladas em uma mesclagem de polifases. Embora as execuções iniciais sejam formadas usando a seleção de substituto, uma árvore de prioridades é usada no lugar de uma estrutura de dados, que fixa o número de registros na memória. Isso porque o PostgreSQL pode lidar com tuplas que variam muito no tamanho e tenta garantir a utilização total do espaço de memória de classificação configurado.
4. **Agregação.** A agregação agrupada no PostgreSQL pode ser baseada em classificação ou baseada em hash. Quando o número estimado de grupos distintos é muito grande, o primeiro é usado; caso contrário, a técnica baseada em hash é preferida.

Triggers e restrições

No PostgreSQL (diferente de alguns sistemas comerciais), recursos do banco de dados ativo como triggers e restrições não são implementados na fase de reescrita. Em vez disso, eles são implementados como parte do executor de consulta. Quando triggers e restrições são registrados pelo usuário, os detalhes são associados à informação de catálogo por relação e índice apropriado. O executor processa uma instrução **update**, **delete** e **insert** gerando repetidamente mudanças de tupla para uma relação. Para cada mudança desse tipo (uma operação update, delete ou insert), o executor explicitamente verifica, dispara e impõe prováveis triggers e restrições, antes ou depois da mudança, conforme a necessidade.

Arquitetura do sistema

A arquitetura do sistema do PostgreSQL segue o modelo de processo por transação. Um site PostgreSQL em execução é

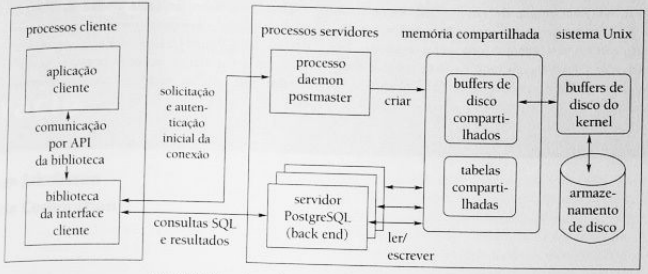


Figura 26.9 A arquitetura do sistema do PostgreSQL.

controlado por um processo coordenador central, chamado postmaster. O processo postmaster é responsável por inicializar e encerrar o servidor e também por tratar de solicitações de conexão de novos clientes. O postmaster atribui cada novo cliente conectando a um processo servidor de back-end que é responsável por executar as consultas em favor do cliente e retornar os resultados ao cliente. Essa arquitetura é representada na Figura 26.9.

Aplicações cliente podem se conectar ao servidor PostgreSQL e submeter consultas por meio de uma das muitas APIs de banco de dados aceitas pelo PostgreSQL (libpq, SQL, ODBC, Perl DBD) que são fornecidas como bibliotecas no cliente. Uma aplicação cliente de exemplo é o programa psql da linha de comandos, incluído na distribuição padrão do PostgreSQL. O postmaster é responsável por lidar com as conexões iniciais do cliente. Para isso, ele escuta constantemente novas conexões sobre uma porta conhecida. Depois de realizar etapas de inicialização, como a autenticação do usuário, o postmaster gerará um novo processo do servidor de back-end para lidar com o novo cliente. Depois dessa conexão inicial, o cliente interage apenas com o processo servidor de back-end, submetendo consultas e recebendo resultados de consulta. Essa é a essência do modelo de processo por conexão adotado pelo PostgreSQL.

O processo servidor de back-end é responsável por executar as consultas submetidas pelo cliente, realizando as etapas necessárias de execução de consulta, incluindo análise, otimização e execução. Cada processo servidor de back-end pode tratar apenas de uma única consulta de cada vez. Para executar mais de uma consulta em paralelo, uma aplicação precisa manter várias conexões com o servidor.

A qualquer momento, pode haver vários clientes conectados ao sistema e, assim, vários processos servidores de back-end podem estar executando ao mesmo tempo. Os processos servidores de back-end acessam dados do banco de dados por meio do pool de buffers da memória principal, que é colocado na memória compartilhada, para que todos os processos tenham a mesma visão dos dados. A memória compartilhada também é usada para implementar outras formas de sincronismo entre os processos servidores, por exemplo, o bloqueio de itens de dados. O uso da memória compartilhada como meio de comunicação requer que um servidor PostgreSQL seja executado em uma única máquina; um site de servidor isolado não pode se espalhar por várias máquinas.

Notas bibliográficas

Existe muita documentação on-line sobre o PostgreSQL em `www.postgresql.org`. Esse site é uma fonte de informações definitiva sobre novas versões do PostgreSQL, que ocorre com freqüência. Até o PostgreSQL versão 8, a única maneira de executar o PostgreSQL sob o Microsoft Windows era usando o Cygwin. Cygwin é um ambiente tipo Linux que permite a recriação de aplicações Linux a partir da origem para execução sob o Windows. Os detalhes estão em `www.cygwin.com`.

Livros sobre PostgreSQL incluem Douglas e Douglas [2003] e Stinson [2002]. Regras conforme utilizadas no PostgreSQL foram apresentadas em Stonebraker *et al.* [1990]. A estrutura GiST é descrita em Hellerstein *et al.* [1995].

As ferramentas de administração do PostgreSQL, pgAccess e pgAdmin são descritas na Web em `www.pgaccess.org` e `www.pgadmin.org`, respectivamente.

As ferramentas de projeto de banco de dados do PostgreSQL, TORA e Data Architect são descritas em `www.globecom.se/tora` e `www.thekompany.com/products/data-architect`, respectivamente.

CAPÍTULO 27

Oracle

Hakan Jakobsson
Oracle Corporation

Quando a Oracle foi fundada em 1977 como Software Development Laboratories por Larry Ellison, Bob Miner e Ed Oates, não havia produtos comerciais de banco de dados relacional. A empresa, que mais tarde passou a se chamar Oracle, partiu para criar um sistema de gerenciamento de banco de dados relacional como um produto comercial, e foi a primeira a chegar ao mercado. Desde então, a Oracle manteve posição de liderança no mercado de banco de dados relacional, mas, com o passar dos anos, seus produtos e ofertas de serviço cresceram além do mercado de servidor de banco de dados relacional. Além das ferramentas relacionadas diretamente ao desenvolvimento e gerenciamento de banco de dados, a Oracle vende ferramentas de inteligência de negócios, incluindo ferramentas de consulta e análise, produtos de mineração de dados (*data mining*) e um servidor de aplicações bastante integrado ao servidor de banco de dados.

Além de servidores e ferramentas relacionadas a banco de dados, a empresa também oferece software de aplicação para planejamento de recursos da empresa e gerenciamento de relacionamento com o cliente, incluindo áreas como finanças, recursos humanos, manufatura, marketing, vendas e gerenciamento da cadeia de suprimentos. A unidade On Demand da Oracle oferece serviços nessas áreas como provedora de serviços de aplicação.

Este capítulo aborda um subconjunto dos recursos, opções e funcionalidade dos produtos Oracle. Novas versões dos produtos estão sendo desenvolvidas continuamente, de modo que todas as descrições de produto estão sujeitas a mudanças. O conjunto de recursos descrito aqui é baseado no primeiro release do Oracle10*g*.

Projeto de banco de dados e ferramentas de consulta

A Oracle oferece uma série de ferramentas para projeto de banco de dados, consulta, geração de relatório e análise de dados, incluindo OLAP.

Ferramentas de projeto de banco de dados

A maior parte das ferramentas de projeto está incluída no Oracle Developer Suite. Esse é um pacote de ferramentas para vários aspectos do desenvolvimento de aplicações, incluindo ferramentas para desenvolvimento de formulários, modelagem de dados, relatórios e consultas. O pacote aceita o padrão UML (ver seção "A Unified Modeling Language (UML)" do Capítulo 6) para modelagem de desenvolvimento. Ele oferece modelagem de classe a fim de gerar código para os componentes de negócios para a Java framework, além da modelagem de atividades para modelagem de fluxo de controle de uso geral. O pacote também admite XML para troca de dados com outras ferramentas de UML.

A principal ferramenta de projeto de banco de dados no pacote é o Oracle Designer, que traduz a lógica de negócios e os fluxos de dados para definições de esquema e scripts procedurais para lógica de aplicação. Ele admite técnicas de modelagem como diagramas ER, engenharia de informação e análise e design de objetos. O Oracle Designer armazena o projeto no Oracle Repository, que serve como um único ponto de metadados para a aplicação. Os metadados podem então ser usados para gerar formulários e relatórios. O Oracle Repository oferece gerenciamento de configuração para objetos de banco de dados, aplicações de formulário, classes Java, arquivos XML e outros tipos de arquivos.

O pacote também contém ferramentas de desenvolvimento de aplicação a fim de gerar formulários, relatórios e ferramentas para diversos aspectos de desenvolvimento baseado em Java e XML, incluindo o JDeveloper, que oferece um ambiente completo para desenvolvimento de ponta a ponta de aplicações J2EE. O componente de inteligência de negócios oferece JavaBeans para funcionalidade analítica, como visualização de dados, consulta e cálculos analíticos.

A Oracle também possui uma ferramenta de desenvolvimento de aplicações para depósito de dados (data warehousing), o Oracle Warehouse Builder. Warehouse Builder é outra ferramenta para projeto e implantação de todos os aspectos de um depósito de dados, incluindo projeto de esquema, mapeamento e transformação de dados, processamento de carga de dados e gerenciamento de metadados. O Oracle Warehouse Builder admite esquemas 3NF e estrela, e também pode importar projetos do Oracle Designer. Essa ferramenta, em conjunto com recursos de banco de dados, como tabelas externas e funções de tabela, normalmente elimina a necessidade de ferramentas de terceiros para extração, transformação e carga.

Ferramentas de consulta

Oracle oferece ferramentas para consulta ocasional, geração de relatórios e análise de dados, incluindo OLAP.

O Oracle Application Server Discoverer é uma ferramenta de publicação Web para consulta ocasional, relatório, análise e publicação Web para usuários finais e analistas de dados. Ele permite que os usuários examinem conjuntos de resultados, dados pivô e armazenem os cálculos como relatórios que podem ser publicados em diversos formatos, como planilhas ou HTML. O Discoverer possui assistentes (wizards) para ajudar os usuários finais a visualizar dados como gráficos. A Oracle tem suporte para um grande conjunto de funções analíticas, como avaliação e movimentação de agregação em SQL. A interface de consulta ocasional do Discoverer pode gerar SQL que tira proveito dessa funcionalidade e pode oferecer aos usuários finais uma funcionalidade analítica muito rica. Como o processamento ocorre no sistema de gerenciamento de banco de dados relacional, o Discoverer normalmente não exige um mecanismo de cálculo complexo no cliente, e existe uma versão do Discoverer que é baseada em navegador.

Variações e extensões da SQL

Oracle tem suporte para todos os recursos básicos da SQL:1999, total ou parcialmente, com algumas pequenas exceções, como tipos de dados distintos. Além disso, Oracle tem suporte para uma grande quantidade de outras construções da linguagem, algumas em conformidade com a SQL:1999, enquanto outras são específicas da Oracle em sintaxe ou funcionalidade. Por exemplo, a Oracle tem suporte para as operações OLAP descritas na seção "Análise de dados e OLAP" do Capítulo 18, incluindo avaliação, agregação móvel, cubo e rollup.

Alguns exemplos de extensões SQL são:

- **connect by**, que é uma forma de travessia de árvore que permite cálculos em estilo de fechamento transitivo em uma única instrução SQL. Essa é uma sintaxe específica do Oracle para um recurso que o Oracle já possui desde a década de 1980.
- **upsert** e **multitable inserts**. A operação upsert combina atualização e inserção, e é útil para mesclar novos dados com dados antigos nas aplicações de depósito de dados. Se uma nova linha tiver o mesmo valor de chave de uma linha antiga, a linha antiga é atualizada (por exemplo, acrescentando os valores de medida da nova linha); caso contrário, a nova linha é inserida na tabela. Inserções de tabela múltipla permitem que várias tabelas sejam atualizadas com base em uma única varredura de novos dados.
- Cláusula **with**, que é descrita na seção "A cláusula with" do Capítulo 3.
- Cláusula **model**, que permite cálculos algébricos de array sobre dados relacionais. Para algumas aplicações, a cláusula model pode ser uma alternativa ao uso de planilhas baseadas no PC.

Recursos do tipo objeto relacional

O Oracle possui bastante suporte para construções objeto relacional, incluindo:

- **Tipos de objeto**. Um modelo de herança única é admitido para hierarquias de tipo.
- **Tipos de coleção**. O Oracle admite **varrays**, que são arrays de tamanho variável, e tabelas aninhadas.
- **Tabelas de objeto**. São usadas para armazenar objetos enquanto oferecem uma visão relacional dos atributos dos objetos.
- **Funções de tabela**. São funções que produzem conjuntos de linhas como saída e podem ser usadas na cláusula **from** de uma consulta. As funções de tabela no Oracle podem ser aninhadas. Se uma função de tabela for usada para expressar alguma forma de transformação de dados, o aninhamento de várias funções permite que múltiplas transformações sejam expressas em uma única instrução.
- **Visões de objeto**. Oferecem uma tabela de objeto virtual dos dados armazenados em uma tabela relacional regular. Elas permitem que os dados sejam acessados ou vistos em um estilo orientado a objeto, mesmo que os da-

dos estejam realmente armazenados em um formato relacional tradicional.

- **Metodos.** Podem ser escritos em PL/SQL, Java ou C.
- **Funções de agregação definidas pelo usuário.** Podem ser usadas em instruções SQL da mesma maneira que as funções internas, como **sum** e **count**.
- **XML como um tipo de dado nativo.** Pode ser usado para armazenar e indexar documentos XML. O Oracle também pode converter automaticamente o resultado de qualquer consulta SQL para XML.

O Oracle possui duas linguagens procedurais principais, PL/SQL e Java. PL/SQL foi a linguagem original do Oracle para procedimentos armazenados, e possui uma sintaxe semelhante à usada na linguagem Ada. Java tem o suporte de uma máquina virtual Java dentro do mecanismo de banco de dados. O Oracle oferece um pacote para encapsular procedimentos, funções e variáveis relacionadas em unidades isoladas. O Oracle admite SQLJ (SQL embutida em Java) e SQL, e também oferece uma ferramenta para gerar definições de classe Java correspondentes a tipos de banco de dados definidos pelo usuário.

OLAP

No passado, o produto OLAP da Oracle foi um servidor de banco de dados multidimensional separado. Agora, o processamento OLAP é feito dentro do banco de dados relacional. Havia muitos motivos para sair de um mecanismo de armazenamento multidimensional separado:

- Um mecanismo relacional pode se expandir para conjuntos de dados muito maiores.
- Um modelo de segurança comum pode ser usado para as aplicações analíticas e o depósito de dados.
- A modelagem multidimensional pode ser integrada à modelagem do depósito de dados.
- O sistema de gerenciamento de banco de dados relacional possui um conjunto maior de recursos e funcionalidade em muitas áreas, como alta disponibilidade, backup e recuperação, e suporte para ferramentas de terceiros.
- Não há necessidade de treinar administradores de banco de dados para dois motores de banco de dados.

O principal desafio com a passagem de um mecanismo de banco de dados multidimensional é oferecer o mesmo desempenho. Um sistema de gerenciamento de banco de dados multidimensional que materializa todo ou grandes partes de um cubo de dados pode oferecer tempos de resposta muito rápidos para muitos cálculos. O Oracle resolveu esse problema de várias maneiras:

- Acrescentou suporte para SQL a uma grande variedade de funções analíticas, incluindo cube, rollup, conjuntos de agrupamento, avaliações, agregação móvel, funções lead e lag, histograma de buckets, regressão linear e desvio-padrão, junto com a capacidade para otimizar a execução dessas funções no mecanismo de banco de dados.
- Estendeu as views materializadas para permitir funções analíticas – em particular, conjuntos de agrupamento. A capacidade de materializar partes do cubo ou todo o cubo é fundamental para o desempenho de um sistema de gerenciamento de banco de dados multidimensional, e as views materializadas dão a um sistema de gerenciamento de banco de dados relacional a capacidade de fazer o mesmo.
- Introduziu *workspaces analíticos*, que armazenam dados no formato multidimensional dentro de uma tabela relacional e possuem métodos associados para operações OLAP, como modelagem, alocação, agregação, previsão e análise hipotética. Um workspace analítico pode ser acessado como uma função de tabela em SQL.

Triggers

O Oracle oferece vários tipos de triggers e várias opções para quando e como elas são invocadas. (Ver na seção "Triggers" do Capítulo 8 uma introdução às triggers em SQL.) Triggers podem ser escritas em PL/SQL, em Java ou como chamadas em C.

Para as triggers que são executadas em instruções DML como insert, update e delete, o Oracle admite triggers de linha e **triggers de instrução**. Triggers de linha são executadas uma vez para cada linha que é afetada (atualizada ou excluída, por exemplo) pela operação DML. Uma trigger de instrução é executada apenas uma vez por instrução. Em cada caso, a trigger pode ser definida como uma trigger *before* ou *after*, dependendo se ela deve ser invocada antes ou depois de a operação DM ser executada.

O Oracle permite a criação de triggers **instead of** para views que não podem estar sujeitas a operações DML. Dependendo da definição da view, pode não ser possível para o Oracle traduzir a instrução DML em uma view para modificações das tabelas básicas sem ambigüidade. Logo, operações DML sobre views estão sujeitas a diversas restrições. Um usuário pode criar uma trigger **instead of** sobre uma view para especificar manualmente quais operações sobre as tabelas de base devem ocorrer em resposta à operação DML na view. O Oracle executa a trigger em vez da operação DML e, portanto, oferece um mecanismo para contornar as restrições sobre operações DML contra views.

O Oracle também possui triggers que são executadas em vários outros eventos, como partida ou desligamento de

banco de dados, mensagens de erro do servidor, logon ou logoff do usuário, e instruções DDL como **create**, **alter** e **drop**.

Armazenamento e indexação

Em jargão do Oracle, um banco de dados consiste em informações armazenadas em arquivos acessadas por meio de uma *instância*, que é uma área de memória compartilhada e um conjunto de processos que interagem com os dados nos arquivos.

Tablespaces

Um banco de dados consiste em uma ou mais unidades lógicas de armazenamento, chamadas **tablespaces**. Cada tablespace, por sua vez, consiste em uma ou mais estruturas chamadas **arquivos de dados**. Estes podem ser arquivos gerenciados pelo sistema operacional ou dispositivos brutos.

Normalmente, um banco de dados Oracle terá as seguintes tablespaces:

- A tablespace **system**, que sempre é criada. Ela contém tabelas e armazenamento do dicionário de dados para triggers e procedimentos armazenados.
- Tablespaces criadas para armazenar dados do usuário. Embora os dados do usuário possam ser armazenados na tablespace **system**, normalmente se deseja separar os dados do usuário dos dados do sistema. Normalmente, a decisão sobre quais outras tablespaces devem ser criadas é baseada no desempenho, disponibilidade, facilidade de manutenção e facilidade de administração. Por exemplo, ter várias tablespaces pode ser útil para operações parciais de backup e recuperação.
- Tablespaces temporárias. Muitas operações de banco de dados exigem a classificação dos dados, e a rotina de classificação pode ter de armazenar dados temporariamente no disco se a classificação não puder ser feita na memória. Tablespaces temporárias são alocadas para classificação e hashing, a fim de tornar mais eficientes as operações de gerenciamento de espaço envolvidas na passagem para o disco.

Tablespaces também podem ser usadas como um meio de mover dados entre bancos de dados. Por exemplo, é comum mover dados de um sistema transacional para um depósito de dados em intervalos regulares. O Oracle permite a movimentação de todos os dados em uma tablespace de um sistema para o outro simplesmente copiando os arquivos e exportando e importando uma pequena quantidade de metadados do dicionário de dados. Essas operações podem ser muito mais rápidas do que descarregar os dados de um banco de dados e depois usar um carregador para inseri-las no outro.

Segmentos

O espaço em uma tablespace é dividido em unidades, chamadas **segmentos**, que contêm dados para uma estrutura de dados específica. Existem quatro tipos de segmentos.

- **Segmentos de dados.** Cada tabela em uma tablespace possui seu próprio segmento em que os dados da tabela são armazenados, a menos que a tabela seja particionada; se for, existe um segmento de dados por partição. (O particionamento no Oracle é descrito na seção "Particionamento".)
- **Segmentos de índice.** Cada índice em uma tablespace tem seu próprio segmento de índice, exceto para índices particionados, que possuem um segmento de índice por partição.
- **Segmentos temporários.** Estes são segmentos usados quando uma operação de classificação precisa gravar dados em disco ou quando os dados são inseridos em uma tabela temporária.
- **Segmentos de rollback.** Estes segmentos contêm informações de undo para que uma transação não confirmada possa ser revertida. Eles também desempenham um papel importante no modelo de controle de concorrência do Oracle e para recuperação de banco de dados, descrito nas seções "Controle de concorrência" e "Estruturas básicas para recuperação".

Abaixo do nível de segmento, o espaço é alocado em um nível de granularidade chamado *extensão*. Cada extensão consiste em um conjunto de *blocos* de banco de dados contíguos. Um bloco de banco de dados é o menor nível de granularidade em que o Oracle realiza E/S de disco. Um bloco de banco de dados não precisa ser igual a um bloco do sistema operacional em tamanho, mas deve ser um múltiplo dele.

O Oracle oferece parâmetros de armazenamento que levam em conta o controle detalhado de como o espaço é alocado e gerenciado:

- O tamanho de uma nova extensão que deve ser alocada de modo a oferecer espaço para linhas que são inseridas em uma tabela.
- A porcentagem de utilização de espaço em que um bloco do banco de dados é considerado cheio e em que linhas não serão mais inseridas. (Deixar algum espaço livre em um bloco pode permitir que linhas existentes cresçam em tamanho por meio de atualizações, sem esgotar o espaço no bloco.)

Tabelas

Uma tabela padrão no Oracle é organizada em heap; ou seja, o local de armazenamento de uma linha em uma tabela não é baseado nos valores contidos na linha, e é fixado quando a linha é inserida. Porém, se a tabela for particionada, o conteúdo da linha afeta a partição em que é armazenada. Existem diversos recursos e variações.

Tabelas de heap podem ser compactadas opcionalmente. O Oracle usa um algoritmo de compactação baseado em dicionário, que é aplicado a cada bloco de dados individualmente. Para tabelas que contêm grandes quantidades de valores repetidos, as economias em espaço de disco e, portanto, E/S de disco, podem ser muito grandes, mas a compactação/descompactação dos dados incorre em uma pequena sobrecarga de CPU.

O Oracle admite tabelas aninhadas; ou seja, uma tabela pode ter uma coluna cujo tipo de dados é outra tabela. A tabela aninhada não é armazenada em linha na tabela pai, mas é armazenada em uma tabela separada.

O Oracle admite tabelas temporárias em que a duração dos dados é a transação em que eles são inseridos ou a sessão do usuário. Os dados são privados à sessão e são removidos automaticamente ao final de sua duração.

Um *cluster* é outra forma de organização para dados de tabela (ver seção "Organização de registros em arquivos" do Capítulo 11). O conceito, nesse contexto, não deve ser confundido com outros significados da palavra *cluster*, como aqueles relacionados à arquitetura de hardware. Em um cluster, as linhas de diferentes tabelas são armazenadas juntas no mesmo bloco com base em algumas colunas comuns. Por exemplo, uma tabela de departamentos e uma tabela de funcionários poderiam ser agrupadas de modo que cada linha na tabela de departamentos fosse armazenada junto com todas as linhas de funcionários para os funcionários que trabalham nesse departamento. Os valores de chave primária/chave estrangeira são usados para determinar o local do armazenamento. Essa organização oferece benefícios de desempenho quando as duas tabelas são juntadas, mas sem a penalidade de espaço de um esquema desnormalizado, pois os valores na tabela de departamentos não são repetidos para cada funcionário. Em contrapartida, uma consulta envolvendo apenas a tabela de departamentos pode ter de envolver um número substancialmente maior de blocos do que se essa tabela tivesse sido armazenada isoladamente.

A organização de cluster implica que uma linha pertence a um local específico; por exemplo, uma nova linha de funcionário precisa ser inserida com as outras linhas para o mesmo departamento. Portanto, um índice sobre a coluna de agrupamento é obrigatório. Uma organização alternativa é um *cluster de hash*. Aqui, o Oracle calcula o local de uma linha aplicando uma função de hash ao valor para a coluna de agrupamento. A função de hash mapeia a linha a um bloco específico no cluster de hash. Como nenhuma travessia de índice é necessária para acessar uma linha de acordo com seu valor de coluna de agrupamento, essa organização pode economizar quantidades significativas de E/S de disco. Porém, o número de buckets de hash e outros parâmetros de armazenamento precisam ser definidos cuidadosamente, para evitar problemas de desempenho em decorrência de muitas colisões ou desperdício de espaço devido a buckets de hash vazios.

As organizações de clusters de hash e clusters regulares podem ser aplicadas a uma única tabela. O armazenamento de uma tabela como um cluster de hash com a coluna de chave primária como chave de cluster pode permitir um acesso baseado em um valor de chave primária com uma única E/S de disco, desde que não haja estouro para esse bloco de dados.

Tabelas organizadas por índice

Em uma tabela *organizada por índice*, os registros são armazenados em um índice de árvore B do Oracle no lugar de uma heap. Uma tabela organizada em índice exige que uma chave exclusiva seja identificada para uso como chave de índice. Embora uma entrada em um índice regular contenha o valor de chave e row-id da linha indexada, uma tabela organizada em índice substitui a row-id por valores de coluna para as colunas restantes da linha. Em comparação com o armazenamento dos dados em uma tabela de heap normal e a criação de um índice sobre as colunas de chave, uma tabela organizada em índice pode melhorar o desempenho e a utilização de espaço. Considere a pesquisa de todos os valores de coluna de uma linha, dado seu valor de chave primária. Para uma tabela de heap, isso exigiria uma sondagem de índice seguida por um acesso de tabela por row-id. Para uma tabela organizada por índice, somente a sondagem de índice é necessária.

Índices secundários sobre colunas não-chave de uma tabela organizada em índice são diferentes de índices sobre uma tabela de heap regular. Em uma tabela de heap, cada linha possui uma row-id fixa, que não muda. Porém, uma árvore binária é reorganizada enquanto cresce ou diminui quando entradas são inseridas ou excluídas, e não existe garantia de que uma linha permanecerá em um local fixo dentro de uma tabela organizada em índice. Logo, um índice secundário sobre uma tabela organizada em índice não contém row-ids normais, mas sim **row-ids lógicas**. Uma row-id lógica consiste em duas partes: uma row-id física, correspondente a onde a linha estava quando o índice foi criado ou recriado por último, e um valor para a chave exclusiva. A row-id física é conhecida como "estimativa", pois poderia ser incorreta se a linha tiver sido movida. Nes-

se caso, a outra parte de uma row-id lógica, o valor de chave para a linha, é usada para acessar a linha; porém, esse acesso é mais lento do que se a estimativa tivesse sido correta, pois envolve uma travessia da árvore binária para a tabela organizada em índice desde a raiz até os nós de folha, potencialmente incorrendo em várias E/Ss de disco. Porém, se uma tabela for altamente volátil e uma grande porcentagem das estimativas tiverem chance de ser erradas, pode ser melhor criar o índice secundário apenas com valores de chave, pois o uso de uma estimativa incorreta pode resultar em uma E/S de disco desperdiçada.

Índices

Oracle admite vários tipos diferentes de índices. O tipo mais usado é o que o Oracle (e vários outros fornecedores) chama de índice de árvore binária (embora seja aquilo que, na verdade, chamamos de índice de árvore B+ no Capítulo 12), criado sobre uma ou várias colunas. As entradas de índice têm o seguinte formato: para um índice sobre as colunas col_1, col_2 e col_3, cada linha na tabela onde pelo menos uma das colunas possui um valor não nulo resultaria na entrada de índice

$$<col_1><col_2><col_3><row\text{-}id>$$

onde $<col_i>$ indica o valor para a coluna i e *<row-id>* é a row-id para a linha. O Oracle opcionalmente pode compactar o prefixo da entrada para economizar espaço. Por exemplo, se houver muitas combinações repetidas de valores $<col_1><col_2>$, a representação de cada prefixo $<col_1><col_2>$ distinto pode ser compartilhada entre as entradas que têm essa combinação de valores, em vez de armazenadas explicitamente para cada entrada desse tipo. A compactação de prefixo pode levar a economias de espaço substanciais.

Índices de mapa de bits

Os índices de mapa de bits (descritos na seção "Índices de mapa de bits" do Capítulo 12) utilizam uma representação de mapa de bits para entradas de índice, o que pode levar a uma economia de espaço substancial (e, portanto, economias de E/S de disco), quando a coluna indexada tiver um número moderado de valores distintos. Os índices de mapa de bits no Oracle utilizam o mesmo tipo de estrutura de árvore binária para armazenar as entradas que um índice regular. Porém, onde um índice regular sobre uma coluna tiver entradas na forma $<col_1><row\text{-}id>$, uma entrada de índice de mapa de bits terá a forma

$$<col_1><row\text{-}id\text{-}inicial><row\text{-}id\text{-}final><mapabitscompact>$$

O mapa de bits conceitualmente representa o espaço de todas as linhas possíveis na tabela entre a row-id inicial e final. O número dessas linhas possíveis em um bloco depende de quantas linhas podem caber em um bloco, que é uma função do número de colunas na tabela e seus tipos de dados. Cada bit no mapa de bits representa uma linha possível em um bloco. Se o valor de coluna dessa linha for aquele da entrada de índice, o bit é definido como 1. Se a linha tiver algum outro valor, ou se ela não existir realmente na tabela, o bit é definido como 0. (É possível que a linha não exista realmente, pois um bloco da tabela pode ter um número menor de linhas que o número que foi calculado como máximo possível.) Se a diferença for grande, o resultado pode ser longas seqüências de zeros consecutivos no mapa de bits, mas o algoritmo de compactação cuida dessas seqüências de zeros, de modo que o efeito negativo é limitado.

O algoritmo de compactação é uma variação de uma técnica de compactação chamada Byte-aligned Bitmap Compression (BBC). Basicamente, uma seção do mapa de bits onde a distância entre dois 1s consecutivos é muito pequena é armazenada como mapas de bits reais. Se a distância entre dois 1s for suficientemente grande – ou seja, se houver um número suficiente de zeros adjacentes entre eles – é armazenado apenas o número de zeros.

Os índices de mapa de bits permitem que vários índices na mesma tabela sejam combinados no mesmo caminho de acesso se houver várias condições sobre colunas indexadas na cláusula where de uma consulta. Por exemplo, para a condição

$$(col_1 = 1 \text{ or } col_1 = 2) \text{ and } col_2 > 5 \text{ and } col_3 < > 10$$

O Oracle poderia calcular quais linhas combinam com a condição realizando operações Booleanas sobre mapas de bits a partir dos índices nas três colunas. Nesse caso, estas operações ocorreriam para cada índice:

- Para o índice sobre col_1, os mapas de bits para os valores de coluna 1 e 2 passariam por um **or** lógico.
- Para o índice sobre col_2, todos os mapas de bits para valores de chave > 5 seriam mesclados em uma operação que corresponde a um **or** lógico.
- Para o índice sobre col_3, os mapas de bits para valores de chave 10 e **null** seriam obtidos. Depois, um **and** Booleano seria realizado sobre os resultados dos dois primeiros índices, seguido por dois menos Booleanos dos mapas de bits para os valores 10 e **null** para col_3.

Todas as operações são realizadas diretamente sobre a representação compactada dos mapas de bits – nenhuma descompactação é necessária –, e o mapa de bits resultante (compactado) representa as linhas que combinam com todas as condições lógicas.

A capacidade de usar as operações Booleanas para combinar com vários índices não está limitada aos índices de

mapa de bits. O Oracle pode converter row-ids em representação de mapa de bits compactada, de modo que pode usar um índice de árvore binária regular em qualquer lugar em uma árvore Booleana de operação do mapa de bits simplesmente colocando um operador row-id para mapa de bits em cima do acesso ao índice no plano de execução.

Via de regra, os índices de mapa de bits costumam economizar mais espaço do que os índices de árvore binária se o número de valores de chave distintos for menor que metade do número de linhas na tabela. Por exemplo, em uma tabela com 1 milhão de linhas, um índice sobre uma coluna com menos de 500.000 valores distintos provavelmente seria menor se fosse criada como um índice de mapa de bits. Para colunas com um número muito pequeno de valores distintos – por exemplo, colunas que se referem a propriedades como país, estado, sexo, estado civil e vários outros flags de status –, um índice de mapa de bits poderia exigir apenas uma pequena fração do espaço de um índice normal de árvore binária. Qualquer vantagem de espaço desse tipo também pode aumentar as vantagens de desempenho correspondentes na forma de menos E/Ss de disco quando o índice for varrido.

Índices baseados em função

Além de criar índices sobre uma ou várias colunas de uma tabela, o Oracle permite que os índices sejam criados sobre expressões que envolvem uma ou mais colunas, como $col_1 + col_2 * 5$. Por exemplo, é possível criar um índice sobre a expressão *upper*(*nome*), onde *upper* é uma função que retorna a versão em maiúsculas de uma string, e *nome* é uma coluna, a fim de realizar pesquisas sem diferenciação de maiúsculas/minúsculas sobre a coluna *nome*. Para encontrar todas as linhas com o nome "van Gogh" de forma eficiente, a condição

upper(*nome*) = 'VAN GOGH'

seria usada na cláusula **where** da consulta. O Oracle, então, combina a condição com a definição de índice e conclui que o índice pode ser usado para apanhar todas as linhas que combinam com "van Gogh", independente de como o nome foi digitado quando armazenado no banco de dados. Um índice baseado em função pode ser criado como um mapa de bits ou como um índice de árvore binária.

Índices de junção

Um índice de junção é um índice em que as colunas de chave não estão na tabela que é referenciada pelas row-ids no índice. O Oracle admite índices de junção de mapa de bits principalmente para uso com esquemas de estrela (ver seção "Esquemas de depósito" do Capítulo 18). Por exemplo, se houver uma coluna para nomes de produto em tabela de dimensão de produto, um índice de junção de mapa de bits sobre a tabela de fatos com essa coluna de chave poderia ser usado para obter as linhas da tabela de fatos que correspondem a um produto com um nome específico, embora o nome não esteja armazenado na tabela de fatos. A forma como as linhas nas tabelas de fatos e dimensões se correspondem é algo baseado em uma condição de junção que é especificada quando o índice é criado, e torna-se parte dos metadados de índice. Quando uma consulta for processada, o otimizador procurará a mesma condição de junção na cláusula **where** da consulta, a fim de determinar se o índice da junção é aplicável.

O Oracle permite que índices de junção de mapa de bits tenham mais de uma coluna de chave, e essas colunas podem estar em diferentes tabelas. Em todos os casos, as condições de junção entre a tabela de fatos em que o índice é baseado e as tabelas de dimensão precisam se referir a chaves exclusivas nas tabelas de dimensão; ou seja, uma linha indexada na tabela de fatos precisa corresponder a uma linha exclusiva em cada uma das tabelas de dimensão.

O Oracle pode combinar um índice de junção de mapa de bits em uma tabela de fatos com outros índices na mesma tabela – sejam índices de junção ou não – usando os operadores para operações de mapa de bits Booleanas. Por exemplo, considere um esquema com uma tabela de fatos para vendas e tabelas de dimensão para clientes, produtos e tempo. Suponha que uma consulta solicite informações sobre vendas para clientes em determinado código postal, que tenham comprado produtos de uma certa categoria durante um certo período de tempo. Se existir um índice de junção de mapa de bits sobre múltiplas colunas, em que as colunas de chave sejam as colunas da tabela de dimensão restrita (código postal, categoria de produto e tempo), o Oracle poderá usar o índice de junção para encontrar linhas na tabela de fatos que combinam com as condições de restrição. Porém, se houver índices individuais, de única coluna, para as colunas de chave (ou um subconjunto delas), o Oracle poderá obter mapas de bits para as linhas da tabela de fatos que combinam com cada condição individual e usar a operação Booleana and para gerar um mapa de bits de tabela de fatos para as linhas que satisfaçam todas as condições. Se a consulta tiver condições sobre algumas das colunas da tabela de fatos, os índices sobre essas colunas poderão ser incluídos no mesmo caminho de acesso, mesmo que sejam índices de árvore binária regulares ou índices de domínio (os índices de domínio são descritos a seguir, na próxima seção).

Índices de domínio

O Oracle permite que as tabelas sejam indexadas por estruturas de índice que não são nativas do Oracle. Esse recurso

de extensibilidade do servidor Oracle permite que os fornecedores de software desenvolvam os chamados **cartuchos**, com funcionalidade para domínios de aplicação específicos, como texto, dados espaciais e imagens, com funcionalidade de indexação além daquela oferecida pelos tipos de índice padrão do Oracle. Na implementação da lógica para criar, manter e pesquisar o índice, o projetista de índice precisa garantir que adere a um protocolo específico em sua interação com o servidor Oracle.

Um índice de domínio precisa ser registrado no dicionário de dados, junto com os operadores que ele admite. O otimizador do Oracle considera índices de domínio como um dos caminhos de acesso possíveis para uma tabela. O Oracle permite que funções de custo sejam registradas com operadores, de modo que o otimizador possa comparar o custo de usar o índice de domínio com aqueles de outros caminhos de acesso.

Por exemplo, um índice de domínio para buscas de texto avançadas pode dar suporte a um operador *contains*. Quando esse operador tiver sido registrado, o índice de domínio será considerado como um caminho de acesso para uma consulta como

```
select *
from funcionários
where contains(currículo, 'LINUX')
```

onde *currículo* é uma coluna de texto na tabela *funcionários*. O índice de domínio pode ser armazenado em um arquivo de dados externo ou dentro de uma tabela organizada por índice do Oracle.

Um índice de domínio pode ser combinado com outros índices (mapa de bits ou árvore binária) no mesmo caminho de acesso, convertendo entre a representação de row-id e mapa de bits e usando operações Booleanas de mapa de bits.

Particionamento

O Oracle admite vários tipos de particionamento horizontal de tabelas e índices, e esse recurso desempenha um papel importante na capacidade do Oracle de admitir bancos de dados muito grandes. A capacidade de particionar uma tabela ou índice possui muitas vantagens:

- Backup e recuperação são mais fáceis e mais rápidos, pois podem ser feitos sobre partições individuais, em vez da tabela como um todo.
- A carga de operações em um ambiente de depósito de dados é menos intrusiva: os dados podem ser acrescentados a uma partição, e depois a partição acrescentada a uma tabela, que é uma operação instantânea. De modo semelhante, o descarte de uma partição com dados obsoletos de uma tabela é muito fácil em um depósito de dados que mantém uma janela rolante de dados históricos.
- O desempenho da consulta se beneficia bastante, pois o otimizador pode reconhecer que somente um subconjunto das partições de uma tabela precisa ser acessado a fim de resolver uma consulta (poda de partição). Além disso, o otimizador pode reconhecer que, em uma junção, não é necessário tentar combinar todas as linhas de uma tabela com todas as linhas da outra, mas que as junções precisam ser feitas apenas entre pares de partições que combinam (junção por partição).

Cada linha em uma tabela particionada está associada a uma partição específica. Essa associação é baseada na coluna ou nas colunas de particionamento que fazem parte da definição de uma tabela particionada. Existem várias maneiras de mapear valores de coluna em partições, fazendo surgir vários tipos de particionamento, cada um com diferentes características: intervalo, hash, lista e particionamento composto.

Particionamento de intervalo

No particionamento de intervalo, os critérios de particionamento são intervalos de valores. Esse tipo de particionamento é especialmente adequado para colunas de data, quando todas as linhas no mesmo intervalo de datas, digamos, um dia ou um mês, pertencem à mesma partição. Em um depósito de dados em que os dados são carregados a partir dos sistemas transacionais em intervalos regulares, o particionamento de intervalo pode ser usado para implementar de modo eficiente uma janela rolante com dados históricos. Cada carga de dados recebe sua própria partição nova, tornando o processo de carregamento mais rápido e mais eficiente. O sistema, na realidade, carrega os dados para uma tabela separada com a mesma definição de coluna da tabela particionada. Ele pode, então, verificar a consistência dos dados, limpá-los e indexá-los. Depois disso, o sistema pode tornar a tabela separada uma nova partição da tabela particionada, por uma simples mudança nos metadados no dicionário de dados – uma operação quase instantânea.

Até a mudança dos metadados, o processo de carga não afeta os dados existentes na tabela particionada de forma alguma. Não é preciso haver qualquer manutenção de índices existentes como parte da carga. Dados antigos podem ser removidos de uma tabela simplesmente descartando sua partição; essa operação não afeta as outras partições.

Além disso, as consultas em um ambiente de depósito de dados normalmente contêm condições que as restringem a um certo período de tempo, como um trimestre ou um

mês. Se for usado o particionamento do intervalo de datas, o otimizador de consulta poderá restringir o acesso aos dados às partições que são relevantes à consulta, evitando uma varredura da tabela inteira.

Particionamento de hash

No particionamento de hash, uma função de hash mapeia linhas em partições, de acordo com os valores nas colunas de particionamento. Esse tipo de particionamento é útil principalmente quando for importante distribuir as linhas por igual entre as partições ou quando as junções de partições forem importantes para o desempenho da consulta.

Particionamento composto

No particionamento composto, a tabela é particionada por intervalo, mas cada partição é subparticionada usando o particionamento de hash ou lista. Esse tipo de particionamento combina as vantagens do particionamento de intervalo e particionamento de hash ou lista.

Particionamento de lista

No particionamento de lista, os valores associados a determinada partição são indicados em uma lista. Esse tipo de particionamento é útil se os dados na coluna de particionamento tiverem um conjunto relativamente pequeno de valores discretos. Por exemplo, uma tabela com uma coluna de estado pode ser particionada implicitamente por região geográfica se cada lista de partição tiver os estados que pertencem à mesma região.

Views materializadas

O recurso de view materializada (ver seção "Definição de view" do Capítulo 3) permite que o resultado de uma consulta SQL seja armazenado em uma tabela e usado para processamento de consulta mais tarde. Além disso, o Oracle mantém o resultado materializado, atualizando-o quando as tabelas que foram referenciadas na consulta forem atualizadas. As views materializadas são usadas no depósito de dados para agilizar o processamento da consulta, mas a tecnologia também é usada para replicação nos ambientes distribuído e móvel.

No depósito de dados, um uso comum para as views materializadas é resumir dados. Por exemplo, um tipo comum de consulta pede "a soma das vendas para cada trimestre durante os últimos dois anos". O cálculo prévio do resultado, ou de algum resultado parcial de tal consulta, pode agilizar bastante o processamento da consulta em comparação com seu cálculo do zero, agregando todos os registros de vendas no nível de detalhe.

O Oracle admite reescritas de consulta automáticas, que tiram proveito de qualquer view materializada útil ao resolver uma consulta. A reescrita consiste em mudar a consulta para usar a view materializada no lugar das tabelas originais na consulta. Além disso, a reescrita pode acrescentar junções adicionais ou agregar processamento, conforme seja necessário, para chegar ao resultado correto. Por exemplo, se uma consulta precisar de vendas por trimestre, a reescrita pode tirar proveito de uma view que materializa vendas por mês, acrescentando uma agregação adicional para converter os meses para trimestres. O Oracle possui um tipo de objeto de metadados chamado **dimension**, que permite que sejam definidos relacionamentos hierárquicos em tabelas. Por exemplo, para uma tabela de dimensão tempo em um esquema de estrela, o Oracle pode definir um objeto de metadados de dimensão a fim de especificar como os dias se transformam em meses, meses em trimestres, trimestres em anos, e assim por diante. De modo semelhante, propriedades hierárquicas relacionadas à geografia podem ser especificadas – por exemplo, como distritos de vendas se transformam em regiões. A lógica de reescrita de consulta examina esses relacionamentos, pois eles permitem que uma view materializada seja usada para classes de consultas mais amplas.

O objeto container para uma view materializada é uma tabela, o que significa que uma view materializada pode ser indexada, particionada ou sujeita a outros controles, para melhorar o desempenho da consulta.

Quando existem mudanças nos dados nas tabelas referenciadas na consulta que define uma view materializada, a view materializada precisa ser renovada para refletir essas mudanças. O Oracle admite a renovação total de uma view materializada e a renovação rápida, incremental. Em uma renovação total, o Oracle recalcula a view materializada do zero, que pode ser a melhor opção se as tabelas básicas tiverem mudanças significativas, por exemplo, mudanças devido a uma carga em massa. Em uma renovação incremental, o Oracle atualiza a view usando os registros que foram alterados nas tabelas básicas; a renovação na view é imediata – ou seja, ela é executada como parte da transação que alterou as tabelas básicas. A renovação incremental pode ser melhor se o número de linhas que foram alteradas for baixo. Existem algumas restrições sobre as classes de consultas para as quais uma view materializada pode ser renovada de modo incremental (e outras para quando uma view materializada pode ser criada).

Uma view materializada é semelhante a um índice no sentido de que, embora possa melhorar o desempenho da consulta, ela utiliza espaço, e sua criação e manutenção consome recursos. Para ajudar a resolver essa decisão, o Oracle oferece um conselheiro que poderá ajudar um usuário a criar as views materializadas mais econômicas, dada uma carga de trabalho de consulta específica como entrada.

Processamento e otimização de consulta

O Oracle admite uma grande variedade de técnicas de processamento em seu mecanismo de processamento de consulta. Algumas das mais importantes são resumidas a seguir.

Métodos de execução

Os dados podem ser acessados por meio de diversos métodos de acesso:

- **Varredura total da tabela**. O processador de consulta varre a tabela inteira, obtendo informações sobre os blocos que compõem a tabela a partir do mapa de extensão e varrendo esses blocos.
- **Varredura de índice**. O processador cria uma chave de início e/ou fim a partir das condições na consulta e a utiliza para varrer para uma parte relevante do índice. Se houver colunas que precisam ser obtidas, e elas não fizerem parte do índice, a varredura de índice é seguida por um acesso de tabela por row-id de índice. Se nenhuma chave de início ou fim estiver disponível, a varredura é de índice completa.
- **Varredura completa rápida de índice**. O processador varre as extensões da mesma maneira que a extensão da tabela em uma varredura de tabela completa. Se o índice tiver todas as colunas da tabela que são necessárias para essa tabela, e não houver boas chaves de início/fim que reduziriam significativamente aquela parte do índice que seria examinada em uma varredura normal do índice, esse método pode ser a forma mais rápida de acessar os dados. Isso porque a varredura completa rápida pode tirar todo o proveito da E/S de disco em múltiplos blocos. Porém, ao contrário de uma varredura completa normal, que atravessa os blocos de folha do índice em ordem, uma varredura completa rápida não garante que a saída preserva a ordem de classificação do índice.
- **Junção de índice**. Se uma consulta só precisar de um pequeno subconjunto das colunas de uma tabela larga, mas nenhum índice isolado tiver todas essas colunas, o processador poderá usar uma junção de índice para gerar as informações relevantes sem acessar a tabela, juntando vários índices que juntos contêm as colunas necessárias. Ela realiza as junções como junções de hash sobre as row-ids dos diferentes índices.
- **Cluster e acesso a cluster de hash**. O processador acessa os dados usando a chave do cluster.

O Oracle tem várias maneiras de combinar informações de vários índices em um único caminho de acesso. Essa capacidade permite que várias condições de cláusula **where** sejam usadas juntas para calcular o conjunto de resultados da forma mais eficiente possível. A funcionalidade inclui a capacidade de realizar operações Booleanas **and**, **or** e **minus** sobre mapas de bits representando row-ids. Há também operadores que mapeiam uma lista de row-ids para mapas de bits e vice-versa, o que permite que índices de árvore binária regulares e índices de mapa de bits sejam usados juntos no mesmo caminho de acesso. Além disso, para muitas consultas envolvendo **count**(*) sobre seleções em uma tabela, o resultado pode ser calculado simplesmente contando-se os bits que são definidos no mapa de bits pela aplicação de condições da cláusula **where**, sem acessar a tabela.

O Oracle admite vários tipos de junções no mecanismo de execução: junções internas, junções externas, semijunções e antijunções. (Uma antijunção no Oracle retorna linhas da entrada do lado esquerdo que não combinam com qualquer linha na entrada do lado direito; essa operação é chamada anti-semijunção em outra literatura.) Ele avalia cada tipo de junção por um destes três métodos: junção de hash, junção sort-merge, ou junção de loop aninhado.

Otimização

No Capítulo 14, discutimos o tópico geral de otimização da consulta. Aqui, discutimos a otimização no contexto do Oracle.

Transformações de consulta

O Oracle realiza otimização de consulta em várias etapas. Uma dessas etapas é realizar diversas transformações de consulta e reescritas que fundamentalmente mudam a estrutura da consulta. Outra etapa é realizar a seleção do caminho de acesso para determinar caminhos de acesso, métodos de junção e ordem de junção. Como algumas transformações nem sempre são benéficas, o Oracle admite transformações baseadas em custo, nas quais as transformações e a seleção do caminho de acesso são intercaladas. Para cada transformação tentada, a seleção do caminho de acesso é realizada a fim de gerar uma estimativa de custo, e a transformação é aceita ou rejeitada com base no custo do plano de execução resultante.

Alguns dos principais tipos de transformações e reescritas aceitos pelo Oracle são os seguintes:

- **Mesclagem de view**. Uma referência de view em uma consulta é substituída pela definição de view. Essa transformação não se aplica a todas as views.
- **Mesclagem de view complexa**. O Oracle oferece esse recurso para certas classes de views que não estão sujeitas à mesclagem de view normal, pois possuem um **group by** ou **select distinct** na definição da view. Se essa view for juntada com outras tabelas, o Oracle poderá comutar

as junções e a operação de sort ou hash usada para o **group by** ou **distinct**.

- **Achatamento de subconsulta.** O Oracle possui uma série de transformações que convertem várias classes das subconsultas para junções, semijunções ou antijunções.
- **Reescrita de view materializada.** O Oracle tem a capacidade de reescrever uma consulta automaticamente para tirar proveito das views materializadas. Se alguma parte da consulta puder ser combinada com uma view materializada existente, o Oracle pode substituir essa parte da consulta por uma referência à tabela em que a view é materializada. Se for necessário, o Oracle acrescenta condições de junção ou operações **group by** para preservar a semântica da consulta. Se diversas views materializadas forem aplicáveis, o Oracle escolhe aquela que oferece a maior vantagem na redução da quantidade de dados que devem ser processados. Além disso, o Oracle sujeita a consulta reescrita e a versão original ao processo de otimização completo, produzindo um plano de execução e uma estimativa de custo associada para cada um. O Oracle, então, decide se deve executar a versão reescrita ou a original da consulta com base nas estimativas de custo.
- **Transformação em estrela.** O Oracle possui uma técnica especial para avaliar consultas contra esquemas em estrela, conhecida como **transformação em estrela**. Quando uma consulta contém uma junção de uma tabela de fatos com tabelas de dimensão, e seleções sobre atributos das tabelas de dimensão, a consulta é transformada pela exclusão da condição de junção entre a tabela de fatos e as tabelas de dimensão, e substituindo a condição de seleção em cada tabela de dimensão por uma subconsulta da forma:

tabela_fatos.fk$_i$ **in**
(**select** *pk* **from** *tabela_dimensão*$_i$
where <condições em *tabela_dimensão*$_i$>)

Uma subconsulta desse tipo é gerada para cada dimensão que possui algum predicado restritivo. Se a dimensão tiver um esquema em floco de neve (ver seção "Depósito de dados" do Capítulo 18), a subconsulta terá uma junção das tabelas apropriadas que compõem a dimensão.

O Oracle usa os valores que são retornados de cada subconsulta para sondar um índice sobre a coluna correspondente na tabela de fatos, obtendo um mapa de bits como resultado. Os mapas de bits gerados de diferentes subconsultas são combinados por uma operação **and** no mapa de bits. O mapa de bits resultante pode ser usado para acessar linhas correspondentes na tabela de fatos. Logo, somente as linhas na tabela de fatos que combinam simultaneamente com as condições nas dimensões restritas serão acessadas.

A decisão sobre o uso de uma subconsulta para determinada dimensão ser econômico e a decisão sobre se a consulta reescrita é melhor do que a original são baseadas nas estimativas de custo do otimizador.

Seleção de caminho de acesso

O Oracle possui um otimizador baseado em custo, que determina a ordem de junção, os métodos de junção e os caminhos de acesso. Cada operação que o otimizador considera possui uma função de custo associada, e o otimizador tenta gerar a combinação de operações que tem o menor custo geral.

Na estimativa do custo de uma operação, o otimizador conta com estatísticas que foram calculadas para objetos de esquema como tabelas e índices. As estatísticas contêm informações sobre o tamanho do objeto, a cardinalidade, a distribuição de dados das colunas de tabela e assim por diante. Para estatísticas de coluna, o Oracle admite histogramas balanceados por altura e de freqüência. Para facilitar a coleta de estatísticas do otimizador, o Oracle pode monitorar a atividade de modificação sobre tabelas e registrar as tabelas que estiveram sujeitas a tantas mudanças que o recálculo das estatísticas pode ser apropriado. O Oracle também acompanha quais colunas são usadas nas cláusulas **where** das consultas, o que as torna candidatas em potencial para a criação de histograma. Com um único comando, um usuário pode dizer ao Oracle para renovar as estatísticas para as tabelas que foram marcadas como suficientemente alteradas. O Oracle usa a amostragem para agilizar o processo de coleta de novas estatísticas e escolhe automaticamente a menor porcentagem de amostra adequada. Ele também determina se a distribuição das colunas marcadas merece a criação de histogramas; se a distribuição estiver perto de uniforme, o Oracle utiliza uma representação mais simples das estatísticas de coluna.

Em alguns casos, pode ser impossível para o otimizador estimar com precisão a seletividade de uma condição na cláusula **where** de uma consulta apenas com base nas estatísticas armazenadas. Por exemplo, a condição pode ser uma expressão envolvendo uma coluna, como f(*col* + 3) > 5. Outra classe de consultas problemáticas são aquelas que possuem vários predicados sobre colunas que possuem alguma forma de correlação. A avaliação da seletividade combinada desses predicados pode ser difícil. O Oracle resolve essas questões por meio da *amostragem dinâmica*. O otimizador pode amostrar aleatoriamente uma pequena parte de uma tabela e aplicar todos os predicados relevantes à amostra para ver se a porcentagem das linhas combina. O Oracle usa custo de CPU e E/Ss de disco no modelo de custo do otimizador. Para equilibrar os dois componentes, ele armazena medidas sobre velocidade de CPU e desempenho de E/S de disco como parte das estatísticas do otimizador. O

pacote do Oracle para colher estatísticas do otimizador calcula essas medidas.

Para consultas envolvendo uma quantidade não trivial de junções, o espaço de busca é um problema para um otimizador de consulta. O Oracle resolve essa questão de várias maneiras. O otimizador gera uma ordem de junção inicial e depois decide sobre os melhores métodos de junção e caminhos de acesso para essa ordem de junção. Depois, ele muda a ordem das tabelas e determina os melhores métodos de junção e caminhos de acesso para a nova ordem de junção, e assim por diante, enquanto mantém o melhor plano que foi encontrado até aqui. O Oracle corta a otimização se o número de ordens de junção diferentes que foram consideradas se tornar tão grande que o tempo gasto no otimizador pode ser observável em comparação com o tempo que seria gasto para executar o melhor plano encontrado até aqui. Como esse corte depende da estimativa de custo para o melhor encontrado até aqui, encontrar logo um bom plano é importante para que a otimização possa ser interrompida após um menor número de ordens de junção, resultando no melhor tempo de resposta. O Oracle usa várias heurísticas de ordenação iniciais para aumentar a probabilidade de que a primeira ordem de junção considerada seja boa.

Para cada ordem de junção que é considerada, o otimizador pode fazer passadas adicionais pelas tabelas a fim de decidir sobre os métodos de junção e caminhos de acesso. Essas passadas adicionais visariam efeitos colaterais globais da seleção do caminho de acesso. Por exemplo, uma combinação específica dos métodos de junção e caminhos de acesso pode eliminar a necessidade de realizar um sort **order by**. Como esse efeito colateral global pode não ser óbvio quando os custos dos diferentes métodos de junção e caminhos de acesso forem considerados localmente, uma passada separada visando um efeito colateral específico é usada para encontrar um possível plano de execução com um melhor custo geral.

SQL Tuning Advisor

Além do processo de otimização regular, o otimizador do Oracle pode ser usado no modo de ajuste como parte do SQL Tuning Advisor, a fim de gerar planos de execução mais eficientes do que normalmente faria. O Oracle monitora a atividade do banco de dados e armazena automaticamente informações sobre instruções SQL de alta carga em um repositório de carga de trabalho. Instruções SQL de alta carga são aquelas que utilizam a maioria dos recursos porque são executados por um grande número de vezes ou porque são inerentemente dispendiosos. O SQL Tuning Advisor pode ser usado para melhorar o desempenho desses sistemas, fazendo vários tipos de recomendações que entram nas diferentes categorias a seguir:

- **Análise de estatísticas.** O Oracle verifica se as estatísticas necessárias pelo otimizador estão faltando ou se estão ultrapassadas e faz recomendações para coletá-las.
- **Profiling SQL.** Um profile para uma instrução SQL é um conjunto de informações que servem para ajudar o otimizador a tomar decisões melhores da próxima vez que a instrução for otimizada. Um otimizador às vezes pode gerar planos de execução ineficientes se for incapaz de estimar cardinalidades e seletividades com precisão, algo que pode acontecer como resultado da correlação de dados ou do uso de certos tipos de construções. Ao executar o otimizador no modo de ajuste para criar um profile, o otimizador tenta verificar se suas suposições estão corretas usando a amostragem dinâmica e a avaliação parcial da instrução SQL. Se encontrar etapas no processo de otimização em que as suposições do otimizador estão erradas, ele gerará um fato de correção para essa etapa, que se tornará parte do profile. A otimização no modo de ajuste pode ser muito demorada, mas pode valer a pena se o uso do profilemelhorar significativamente o desempenho da instrução. Se um profile for criado, ele será armazenado persistentemente e usado sempre que a instrução for otimizada no futuro. Os perfis podem ser usados para ajustar instruções SQL sem mudar o texto da instrução, algo que é importante porque normalmente é impossível que o administrador de banco de dados modifique instruções geradas por uma aplicação.
- **Análise do caminho de acesso.** Com base em uma análise feita pelo otimizador, o Oracle sugere a criação de índices adicionais que poderiam agilizar a instrução.
- **Análise de estrutura SQL.** O Oracle sugere mudanças na estrutura da instrução SQL que levariam em conta a execução mais eficiente.

Para tabelas particionadas, o otimizador tenta corresponder condições na cláusula **where** de uma consulta com os critérios de particionamento para a tabela, a fim de evitar o acesso a partições que não são necessárias para o resultado. Por exemplo, se uma tabela for particionada por intervalo de datas e a consulta for restringida a dados entre duas datas específicas, o otimizador determina quais partições contêm os dados entre as datas especificadas e garante que somente essas partições são acessadas. O cenário é muito comum, e o ganho de velocidade pode ser muito grande se somente um pequeno subconjunto das partições for necessário.

Execução paralela

O Oracle permite que a execução de uma única instrução SQL seja feita em paralelo, dividindo-se o trabalho entre vários processos em um computador multiprocessador. Esse

recurso é especialmente útil para operações computacionalmente intensivas, que de outra forma levariam um tempo inaceitavelmente longo para realizar. Alguns exemplos representativos são consultas de apoio à decisão que precisam processar grandes quantidades de dados, cargas de dados em um depósito de dados e criação ou recriação de índice.

Para conseguir um bom ganho de velocidade pelo paralelismo, é importante que o trabalho envolvido na execução da instrução seja dividido em grânulos que possam ser processados independentemente pelos diferentes processadores paralelos. Dependendo do tipo de operação, o Oracle possui várias maneiras de distribuir o trabalho.

Para operações que acessam objetos de base (tabelas e índices), o Oracle pode dividir o trabalho por fatias horizontais dos dados. Para algumas operações, como varredura total de tabela, cada fatia dessas pode ser um intervalo de blocos – cada processo de consulta paralelo varre a tabela a partir do bloco no início do intervalo até o bloco no final. Para outras operações em uma tabela particionada, como update e delete, a fatia seria uma partição. Para inserções em uma tabela não particionada, os dados a serem inseridos são divididos aleatoriamente pelos processos paralelos.

As junções podem ser colocadas em paralelo de várias maneiras diferentes. Uma delas é dividir uma das entradas para a junção; esse é o método assimétrico fragmentar e replicar da seção "Junção fragmentar-e-replicar" do Capítulo 21. Por exemplo, se uma grande tabela for juntada a uma pequena por uma junção de hash, o Oracle divide a tabela grande entre os processos e envia uma cópia da tabela pequena para cada processo, que depois junta sua fatia com a tabela menor. Se as duas tabelas forem grandes, seria extremamente dispendioso enviar uma delas a todos os processos. Nesse caso, o Oracle consegue o paralelismo particionando os dados entre processos usando o hashing sobre os valores das colunas de junção (o método de junção de hash particionado da seção "Junção particionada" do Capítulo 21). Cada tabela é varrida em paralelo por um conjunto de processos, e cada linha na saída é passada adiante para um dentre um conjunto de processos que devem realizar a junção. Qual desses processos recebe a linha é algo determinado por uma função de hash sobre os valores da coluna de junção. Logo, cada processo de junção obtém apenas as linhas que potencialmente poderiam combinar, e nenhuma linha que pudesse combinar poderia acabar em diferentes processos.

O Oracle coloca em paralelo as operações de classificação por intervalos de valor da coluna em que a classificação é realizada (ou seja, usando a classificação de particionamento de intervalo da seção "Classificação paralela" do Capítulo 21). Cada processo participante da classificação recebe e classifica linhas com valores em seu intervalo. Para maximizar os benefícios do paralelismo, as linhas precisam ser divididas o mais uniformemente possível entre os processos paralelos, e então surge o problema de determinar limites de intervalo que gerem uma boa distribuição. O Oracle resolve o problema com uma amostragem dinâmica de um subconjunto das linhas na entrada da classificação antes de decidir sobre os limites de intervalo.

Os processos envolvidos na execução paralela de uma instrução SQL consistem em um processo coordenador e uma série de processos servidores paralelos. O coordenador é responsável por atribuir trabalho aos servidores paralelos e coletar e retornar dados para o processo do usuário que emitiu a instrução. O grau de paralelismo é o número de processos servidores paralelos que são atribuídos para executar uma operação primitiva como parte da instrução. O grau de paralelismo é determinado pelo otimizador, mas pode ser desacelerado dinamicamente se a carga sobre o sistema aumentar.

Os servidores paralelos operam sobre um modelo de produtor/consumidor. Quando uma seqüência de operações é necessária para processar uma instrução, o conjunto produtor de servidores realiza a primeira operação e passa os dados resultantes ao conjunto consumidor. Por exemplo, se uma varredura de tabela completa for seguida por uma classificação e o grau de paralelismo for 12, haveria 12 servidores produtores realizando a varredura de tabela e passando o resultado para 12 servidores consumidores que realizam a classificação. Se uma operação subseqüente for necessária, como outra classificação, os papéis dos dois conjuntos de servidores são invertidos. Os servidores que originalmente realizavam a varredura de tabela assumem o papel de consumidores da saída produzida pela primeira classificação e a utilizam para realizar a segunda classificação. Logo, uma seqüência de operações prossegue passando dados entre dois conjuntos de servidores que alternam em seus papéis como produtores e consumidores. Os servidores se comunicam entre si por meio de buffers de memória no hardware de memória compartilhada e por meio de conexões de rede de alta velocidade nas configurações MPP (nada compartilhado) e sistemas agrupados (disco compartilhado).

Para sistemas de nada compartilhado, o custo de acessar os dados no disco não é uniforme entre os processos. Um processo sendo executado em um nó que possui acesso direto a um dispositivo é capaz de processar dados sobre esse dispositivo mais rapidamente do que um processo que precisa apanhar os dados por uma rede. O Oracle utiliza o conhecimento sobre a afinidade dispositivo-para-nó e dispositivo-para-processo – ou seja, a capacidade de acessar dispositivos diretamente – ao distribuir o trabalho entre servidores de execução paralela.

Controle de concorrência e recuperação

O Oracle admite técnicas de controle de concorrência e recuperação que oferecem uma série de recursos úteis.

Controle de concorrência

O controle de concorrência de versão múltipla do Oracle difere dos mecanismos de concorrência utilizados pela maioria dos outros fornecedores de banco de dados. As consultas somente leitura recebem um instantâneo com leitura consistente, que é uma view do banco de dados conforme ele existia em um ponto específico do tempo, contendo todas as atualizações que foram confirmadas até esse ponto no tempo, e não contendo quaisquer atualizações que não foram confirmadas nesse ponto no tempo. Assim, bloqueios de leitura não são usados, e consultas somente leitura não interferem com outra atividade do banco de dados em termos de bloqueio. (Esse é basicamente o protocolo de bloqueio em duas fases multiversão, descrito na seção "Bloqueio de duas fases em múltipla versão" do Capítulo 16.)

O Oracle admite consistência de leitura em nível de instrução e transação: no início da execução de uma instrução ou uma transação (dependendo do nível de consistência usado), o Oracle determina o número de mudança do sistema (SCN) atual. O SCN basicamente atua como uma timestamp, em que o tempo é medido em termos de confirmações de transação, em vez de tempo do relógio.

Se, no decorrer de uma consulta, um bloco de dados for encontrado com SCN mais alto do que aquele sendo associado à consulta, fica evidente que o bloco de dados foi modificado após o momento do SCN da consulta original por alguma outra transação que pode ou não ter sido confirmada. Logo, os dados no bloco não podem ser incluídos em uma view consistente do banco de dados conforme existia no momento do SCN da consulta. Em vez disso, é preciso utilizar uma versão mais antiga dos dados no bloco – especificamente, aquela que tem o SCN mais alto que não ultrapasse o SCN da consulta. O Oracle recupera essa versão dos dados a partir do segmento de rollback (segmentos de rollback são descritos na próxima seção). Logo, desde que o segmento de rollback seja suficientemente grande, o Oracle pode retornar um resultado consistente da consulta mesmo que os itens de dados tenham sido modificados várias vezes desde que a consulta iniciou sua execução. Se o bloco com o SCN desejado não existir mais no segmento de rollback, a consulta retornará um erro. Isso seria uma indicação de que o segmento de rollback não foi devidamente dimensionado, dada a atividade no sistema.

No modelo de concorrência do Oracle, operações de leitura não bloqueiam operações de escrita, e operações de escrita não bloqueiam operações de leitura, uma propriedade que permite um alto grau de concorrência. Em particular, o esquema permite que consultas de longa duração (por exemplo, consultas de relatório) sejam executadas em um sistema com uma grande quantidade de atividade transacional. Esse tipo de cenário normalmente é problemático para sistemas de banco de dados em que as consultas são bloqueios de leitura, pois a consulta pode deixar de adquiri-los ou bloquear grandes quantidades de dados por um longo tempo, evitando assim a atividade transacional sobre esses dados e reduzindo a concorrência. (Uma alternativa utilizada em alguns sistemas é usar um grau de consistência menor, como a consistência de grau dois, mas que poderia ocasionar resultados de consulta inconsistentes.)

O modelo de concorrência do Oracle é usado como uma base para os recursos de *Flashback*. Esses recursos permitem que um usuário defina um certo número SCN ou hora do relógio em uma sessão e realizam operações sobre os dados que existiam nesse ponto do tempo (desde que os dados ainda existam no segmento de rollback). Normalmente em um sistema de banco de dados, quando uma mudança tiver sido confirmada, não haverá como retornar ao estado anterior dos dados além de realizar a recuperação do ponto no tempo a partir de backups. Contudo, a recuperação de um banco de dados muito grande pode ser muito dispendiosa, especialmente se o objetivo for apenas recuperar algum item de dados que foi inadvertidamente excluído por um usuário. O recurso Flashback Query oferece um mecanismo muito mais simples para lidar com erros do usuário. Os recursos incluem a capacidade de restaurar uma tabela ou um banco de dados inteiro para um ponto anterior no tempo sem recuperar-se de backups, a capacidade de realizar consultas sobre os dados conforme os dados existiam em um ponto anterior no tempo, a capacidade de acompanhar como uma ou mais linhas mudaram com o tempo, e a capacidade de examinar mudanças no banco de dados no nível de transação.

O Oracle admite dois níveis de isolamento ANSI/ISO, "leitura confirmada" e "seriável". Não há suporte para leituras sujas, pois isso não é necessário. Os dois níveis de isolamento correspondem ao uso de consistência de leitura em nível de instrução ou em nível de transação. A consistência de leitura em nível de instrução é o padrão.

O Oracle usa o bloqueio em nível de linha. As atualizações para diferentes linhas não entram em conflito. Se duas escritas tentarem modificar a mesma linha, uma espera até que a outra confirme ou seja revertida, e depois pode retornar um erro de conflito de escrita ou seguir em frente e modificar a linha. Os bloqueios são mantidos pela duração de uma transação.

Além dos bloqueios em nível de linha que impedem inconsistências devido à atividade de DML, o Oracle usa bloqueios de tabela que impedem inconsistências devido à ati-

vidade de DDL. Esses bloqueios impedem um usuário, digamos, de descartar uma tabela enquanto outro usuário tem uma transação não confirmada que está acessando essa tabela. Para seu controle de concorrência normal, o Oracle não usa escalada de bloqueio a fim de converter bloqueios de linha em bloqueios de tabela.

O Oracle detecta impasses automaticamente e os resolve revertendo uma das transações envolvidas no impasse.

O Oracle admite transações autônomas, que são transações independentes, geradas dentro de outras transações. Quando o Oracle invoca uma transação autônoma, ele gera uma nova transação em um contexto separado. A nova transação pode ser confirmada ou revertida antes que o controle retorne à transação que chama. O Oracle admite vários níveis de aninhamento de transações autônomas.

Estruturas básicas para recuperação

Para entender o modo como o Oracle se recupera de uma falha, como uma falha de disco, é importante entender as estruturas básicas que estão envolvidas. Além dos arquivos de dados que contêm tabelas e índices, existem arquivos de controle, logs de redo, logs de redo arquivados e segmentos de rollback.

O arquivo de controle contém diversos metadados que são necessários para operar o banco de dados, incluindo informações sobre backups.

O Oracle registra qualquer modificação transacional de um buffer de banco de dados no log de redo, que consiste em dois ou mais arquivos. Ele registra a modificação como parte da operação que a causa, independente de se a transação por fim é confirmada. Ele registra as mudanças nos índices e segmentos de rollback, além de mudanças nos dados da tabela. À medida que os logs de redo se enchem, eles são arquivados por um dos vários processos de segundo plano (se o banco de dados estiver rodando no modo **archivelog**).

O segmento de rollback contém informações sobre versões mais antigas dos dados (ou seja, informações de undo). Além do seu papel no modelo de consistência do Oracle, a informação é usada para restaurar a versão antiga dos itens de dados quando uma transação que modificou os itens de dados é revertida.

Para poder recuperar-se de uma falha de armazenamento, os arquivos de dados e arquivos de controle deverão ter backup regular. A freqüência do backup determina o tempo de recuperação no pior caso, pois a recuperação leva mais tempo se o backup for antigo. O Oracle admite backups quentes – backups realizados sobre um banco de dados on-line que está sujeito à atividade transacional.

Durante a recuperação de um backup, o Oracle realiza duas etapas para alcançar um estado consistente do backup conforme existia antes da falha. Primeiro, ele se move para frente aplicando os logs de redo (arquivados) ao backup. Essa ação leva o banco de dados a um estado que existia no momento da falha, mas não necessariamente um estado consistente, pois os logs de redo incluem dados não confirmados. Em segundo lugar, o Oracle reverte transações não confirmadas usando o segmento de rollback. O banco de dados agora está em um estado consistente.

A recuperação sobre um banco de dados que esteve sujeito à atividade transacional pesada desde o último backup pode ser demorada. O Oracle admite a recuperação paralela em que vários processos são usados para aplicar informações de redo simultaneamente. O Oracle oferece uma ferramenta GUI, o **Recovery Manager**, que automatiza a maioria das tarefas associadas ao backup e recuperação.

Oracle Data Guard

Para garantir a alta disponibilidade, o Oracle oferece um recurso de banco de dados de standby, o *Data Guard*. (Esse recurso é o mesmo que os backups remotos, descritos na seção "Sistemas de backup remoto" do Capítulo 17.) Um banco de dados de standby é uma cópia do banco de dados regular que está instalado em um sistema separado. Se houver uma falha catastrófica no sistema primário, o sistema de standby é ativado e assume, minimizando assim o efeito da falha na disponibilidade. O Oracle mantém o banco de dados de standby atualizado aplicando constantemente logs de redo arquivados que são entregues a partir do banco de dados primário. O banco de dados de backup pode ser colocado on-line no modo somente leitura e usado para relatórios e consultas de apoio à decisão.

Arquitetura do sistema

Sempre que uma aplicação de banco de dados executar uma instrução SQL, haverá um processo do sistema operacional que executará código no servidor de banco de dados. O Oracle pode ser configurado de modo que o processo do sistema operacional seja *dedicado* exclusivamente à instrução que está processando ou de modo que o processo possa ser compartilhado entre várias instruções. A última configuração, conhecida como *servidor compartilhado*, possui propriedades um tanto diferentes com relação ao processo e arquitetura de memória. Discutiremos primeiro a arquitetura de servidor dedicado e depois a arquitetura de servidor multithreaded.

Servidor dedicado: estruturas de memória

A memória usada pelo Oracle pode ser principalmente de três categorias: áreas de código de software, que são as partes da memória em que reside o código do servidor Oracle,

a área global do sistema (SGA – System Global Area) e a área global do programa (PGA – Program Global Area).

Uma PGA é alocada a cada processo para manter seus dados locais e informações de controle. Essa área contém espaço de pilha para vários dados de sessão e a memória privada para a instrução SQL que está sendo executada. Ela também contém memória para operações de classificação e hashing que podem ocorrer durante a avaliação da instrução. O desempenho de tais operações é sensível à quantidade de memória disponível. Por exemplo, uma junção de hash que pode ser realizada na memória será mais rápida se for necessário passar para o disco. Como pode haver muitas operações de classificação e hashing ativas simultaneamente (devido a múltiplas consultas e também múltiplas operações dentro de cada consulta), a decisão de quanta memória deve ser alocada a cada operação não é trivial, especialmente quando a carga no sistema puder flutuar. O Oracle permite que o administrador de banco de dados especifique um parâmetro para a quantidade total de memória que deve ser considerada disponível para essas operações e decide dinamicamente a melhor maneira de dividir a memória disponível entre as operações ativas, a fim de aumentar a vazão. O algoritmo de alocação de memória conhece o relacionamento entre a memória e o desempenho para as diferentes operações, buscando garantir que a memória disponível seja usada da forma mais eficiente possível.

A SGA é uma área da memória para estruturas que são compartilhadas entre os usuários. Ela é composta de várias estruturas importantes, incluindo:

- **O cache de buffer.** Esse cache mantém blocos de dados acessados com freqüência (de tabelas e também de índices) na memória, para reduzir a necessidade de realizar a E/S física do disco. Uma política de substituição de uso menos recente é usada, exceto para blocos acessados durante uma varredura completa da tabela. Porém, o Oracle permite que vários pools de buffer sejam criados com diferentes critérios para envelhecimento de dados. Algumas operações do Oracle evitam o cache de buffer e lêem dados diretamente do disco.
- **O buffer de log de redo.** Esse buffer contém a parte do log de redo que ainda não foi gravada em disco.
- **O pool compartilhado.** O Oracle procura maximizar o número de usuários que podem usar o banco de dados simultaneamente, reduzindo a quantidade de memória necessária para cada usuário. Um conceito importante nesse contexto é a capacidade de compartilhar a representação interna das instruções SQL e o código procedural escrito em PL/SQL. Quando vários usuários executam a mesma instrução SQL, eles podem compartilhar a maioria das estruturas de dados que representam o plano de execução para a instrução. Somente os dados que são locais a cada invocação específica da instrução precisam ser mantidos na memória privada.

  As partes compartilháveis das estruturas de dados representando a instrução SQL são armazenadas no pool compartilhado, incluindo o texto da instrução. O caching de instruções SQL no pool compartilhado também economiza tempo de compilação, pois uma nova invocação de uma instrução que já está em cache não precisa passar pelo processo de compilação completo. A determinação de se uma instrução SQL é a mesma existente no pool compartilhado é baseada na combinação de texto exata e na definição de certos parâmetros da sessão. O Oracle pode substituir automaticamente constantes em uma instrução SQL com variáveis de ligação; portanto, consultas futuras que são iguais, exceto pelos valores de constantes, combinarão com a consulta anterior no pool compartilhado. O pool compartilhado também contém caches para informações de dicionário e diversas estruturas de controle.

Servidor dedicado: estruturas de processo

Existem dois tipos de processos que executam o código do servidor Oracle: processos servidores que processam instruções SQL e processos de segundo plano que realizam diversas tarefas administrativas e relacionadas a desempenho. Alguns desses processos são opcionais e, em alguns casos, vários processos do mesmo tipo podem ser usados por motivos de desempenho. Alguns dos tipos mais importantes de processos de segundo plano são:

- **Escritor de banco de dados.** Quando um buffer é removido do cache de buffer, ele precisa ser escrito de volta ao disco se tiver sido modificado desde que entrou no cache. Essa tarefa é realizada pelos processos escritores de banco de dados, que ajudam o desempenho do sistema, liberando espaço no cache de buffer.
- **Escritor de log.** O processo escritor de log escreve as entradas do buffer do log de redo no arquivo de log de redo no disco. Ele também escreve um registro de commit no disco sempre que uma transação é confirmada.
- **Ponto de verificação.** O processo do ponto de verificação atualiza os cabeçalhos do arquivo de dados quando ocorre um ponto de verificação.
- **Monitor do sistema.** Esse processo realiza recuperação de falhas se for necessário. Ele também realiza algum gerenciamento de espaço para reclamar o espaço não usado nos segmentos temporários.
- **Monitor do processo.** Esse processo realiza recuperação de processo para processos servidores que falham, liberando recursos e realizando várias operações de limpeza.

- **Recuperador.** O processo recuperador resolve falhas e realiza limpeza para transações distribuídas.
- **Arquivador.** O arquivador copia o arquivo do log de redo on-line para um log de redo arquivado toda vez que o arquivo de log on-line se enche.

Servidor compartilhado

A configuração de servidor compartilhado aumenta o número de usuários que determinado número de processos servidores pode admitir compartilhando processos servidores entre as instruções. Ela difere da arquitetura de servidor dedicado nestes aspectos importantes:

- Um processo de despacho em segundo plano direciona as solicitações do usuário para o próximo processo servidor disponível. Ao fazer isso, ele usa uma fila de solicitação e uma fila de resposta na SGA. O despachante coloca uma nova solicitação na fila de solicitação, em que será apanhada por um processo servidor. Quando um processo servidor completa uma solicitação, ele coloca o resultado na fila de resposta para ser obtido pelo despachante e retornado ao usuário.
- Como um processo servidor é compartilhado entre várias instruções SQL, o Oracle não mantém dados privados na PGA. Em vez disso, ele armazena os dados específicos da sessão na SGA.

Oracle Real Application Clusters

Oracle Real Application Clusters (RAC) é um recurso que permite que várias instâncias do Oracle sejam executadas sobre o mesmo banco de dados. (Lembre-se de que, na terminologia do Oracle, uma instância é a combinação de processos de segundo plano e áreas da memória.) Esse recurso permite que o Oracle seja executado em arquiteturas de hardware com clusters e MPP (disco compartilhado e nada compartilhado). A capacidade de agrupar vários nós possui benefícios importantes para escalabilidade e disponibilidade, que são úteis em ambientes de OLTP e depósito de dados.

Os benefícios de escalabilidade do recurso são óbvios, pois mais nós significam mais poder de processamento. Em arquiteturas nada compartilhado, a inclusão de nós a um cluster normalmente exige redistribuição dos dados entre os nós. O Oracle otimiza ainda mais o uso do hardware por meio de recursos como afinidade e junções de partição.

O RAC também pode ser usado para conseguir alta disponibilidade. Se um nó falhar, os nós restantes ainda estão disponíveis para a aplicação que acessa o banco de dados. As instâncias restantes automaticamente reverterão transações não confirmadas que estavam sendo processadas no nó que falhou, a fim de impedir que bloqueiem a atividade nos nós restantes.

Ter várias instâncias executando sobre o mesmo banco de dados dá lugar a algumas questões técnicas que não existem em uma única instância. Embora às vezes seja possível particionar uma aplicação entre os nós, para que eles raramente acessem os mesmos dados, sempre há a possibilidade de sobreposições, o que afeta o gerenciamento de cache. Para resolver essa questão, o Oracle usa o recurso de *fusão de cache*, que permite que os blocos de dados fluam diretamente entre os caches em diferentes instâncias usando a interconexão, sem serem gravados em disco.

Replicação, distribuição e dados externos

O Oracle oferece suporte para replicação e transações distribuídas com commit de duas fases.

Replicação

O Oracle admite vários tipos de replicação. (Veja uma introdução à replicação na seção "Replicação de dados" do Capítulo 22.) Em sua forma mais simples, os dados em um site mestre são replicados para outros sites na forma de **snapshots**. (O termo **snapshot** nesse contexto não deve ser confundido com o conceito de um snapshot com leitura consistente no contexto do modelo de concorrência.) Um snapshot não precisa conter todos os dados mestres – ele pode, por exemplo, excluir certas colunas de uma tabela por motivos de segurança. O Oracle admite dois tipos de snapshots: *somente leitura* e *atualizáveis*. Um snapshot atualizável pode ser modificado em um site escravo e as modificações, propagadas na tabela mestre. Porém, os snapshots somente leitura permitem uma maior variedade de definições de snapshot. Por exemplo, um snapshot somente leitura pode ser definido em termos de operações de conjunto sobre tabelas no site mestre.

O Oracle também admite vários sites mestre para os mesmos dados, em que todos os sites mestre atuam como pares. Uma tabela replicada pode ser atualizada em qualquer um dos sites mestres, e a atualização é propagada nos outros sites. As atualizações podem ser propagadas de forma assíncrona ou síncrona.

Para a replicação assíncrona, a informação de atualização é enviada em lotes para os outros sites mestre e aplicada. Como os mesmos dados poderiam estar sujeitos a modificações em conflito em diferentes sites, a solução de conflito baseada em algumas regras comerciais poderia ser necessária. O Oracle oferece uma série de métodos de resolução de conflito embutidos e permite que os usuários escrevam seus próprios métodos, se for necessário.

Com a replicação síncrona, uma atualização em um site mestre é propagada imediatamente para todos os outros sites. Se a transação de atualização falhar em qualquer site mestre, a atualização é descartada em todos os sites.

Bancos de dados distribuídos

O Oracle admite consultas e transações que se espalham por vários bancos de dados em diferentes sistemas. Com o uso de gateways, os sistemas remotos podem incluir bancos de dados não Oracle. O Oracle possui capacidade embutida para otimizar uma consulta que inclui tabelas em diferentes sites, recuperar os dados relevantes e retornar o resultado como se ela tivesse sido uma consulta normal, local. O Oracle também admite transparentemente transações que se espalham por vários sites por um protocolo interno de commit de duas fases.

Fontes de dados externas

O Oracle tem vários mecanismos para dar suporte a fontes de dados externas. O uso mais comum é no depósito de dados, quando grandes quantidades de dados são regularmente carregadas de um sistema transacional.

SQL*Loader

O Oracle possui um utilitário de carga direta, SQL*Loader, que admite cargas paralelas rápidas de grandes quantidades de dados a partir de arquivos externos. Ele admite uma série de formatos de dados e pode realizar várias operações de filtragem sobre os dados sendo carregados.

Tabelas externas

O Oracle permite que fontes de dados externas, como arquivos comuns, sejam referenciadas na cláusula **from** de uma consulta como se fossem tabelas normais. Uma tabela externa é definida por metadados que descrevem os tipos de coluna do Oracle e o mapeamento dos dados externos para essas colunas. Um driver de acesso também é necessário para acessar os dados externos. O Oracle oferece um driver-padrão para arquivos comuns.

O recurso de tabela externa serve principalmente para operações de extração, transformação e carga (ETL – Extraction, Transformation, Loading) em um ambiente de depósito de dados. Os dados podem ser carregados para o depósito de dados a partir de um arquivo comum usando

```
create table tabela as
    select ... from < tabela externa >
    where ...
```

Acrescentando operações sobre os dados na lista de select ou na cláusula where, as transformações e a filtragem podem ser feitas como parte da mesma instrução SQL. Como essas operações podem ser expressas na SQL nativa ou em funções escritas na PL/SQL ou Java, o recurso de tabela externa oferece um mecanismo muito poderoso para expressar todos os tipos de operações de transformação e filtragem de dados. Para a escalabilidade, o acesso à tabela externa pode ser colocado em paralelo pelo recurso de execução paralela do Oracle.

Ferramentas de administração de banco de dados

O Oracle oferece aos usuários uma série de ferramentas e recursos para gerenciamento do sistema e desenvolvimento de aplicações. Na versão Oracle10g, muita ênfase foi dada ao conceito de *facilidade de manuseio*, ou seja, reduzir a complexidade de todos os aspectos da criação e administração de um banco de dados Oracle. Esse esforço cobriu diversas áreas, incluindo criação de banco de dados, ajuste, gerenciamento de espaço, gerenciamento de armazenamento, backup e recuperação, gerenciamento de memória, diagnósticos de desempenho e gerenciamento de carga de trabalho.

Repositório automático de carga de trabalho

O Automatic Workload Repository (AWR) é uma das partes centrais da infra-estrutura para o esforço de facilidade de manuseio do Oracle. O Oracle monitora a atividade no sistema de banco de dados e registra uma série de informações relacionadas a cargas de trabalho e consumo de recursos. A informação registrada é usada para diagnóstico de desempenho e oferece uma base para uma série de *conselheiros* que oferecem análise de vários aspectos de desempenho do sistema e conselhos de como ele pode ser melhorado. O Oracle possui conselheiros para ajuste de SQL, criação de estruturas de acesso, como índices e views materializadas, e dimensionamento de memória. O Oracle também oferece conselheiros para desfragmentação de segmento e dimensionamento de undo.

Gerenciamento de recursos de banco de dados

Um administrador de banco de dados precisa ser capaz de controlar como o poder de processamento do hardware é dividido entre os usuários ou grupos de usuários. Alguns grupos podem executar consultas interativas em que o tempo de resposta é crítico; outros podem executar relatórios de longa execução que podem ser executados como tarefas

em lote em segundo plano quando a carga do sistema for baixa. Também é importante ser capaz de impedir que um usuário inadvertidamente submeta uma consulta ocasional extremamente dispendiosa, que atrasará outros usuários indevidamente.

O recurso Database Resource Management do Oracle permite que o administrador de banco de dados divida os usuários em grupos consumidores de recursos com diferentes prioridades e propriedades. Por exemplo, um grupo de usuários de alta prioridade e interativos podem ter garantias de pelo menos 60% da CPU. O restante, mais qualquer parte dos 60% não usados pelo grupo de alta prioridade, seria alocado entre os grupos consumidores de recursos com menor prioridade. Um grupo com prioridade realmente baixa poderia receber 0%, o que significaria que as consultas emitidas por esse grupo seriam executadas apenas quando houvesse ciclos de CPU disponíveis. Os limites sobre o grau de paralelismo para execução paralela podem ser definidos para cada grupo. O administrador de banco de dados também pode definir limites sobre quanto tempo uma instrução SQL tem permissão para execução para cada grupo. Quando um usuário submete uma instrução, o Resource Manager estima quanto tempo levaria para executá-la e retorna um erro se a instrução violar o limite. O gerenciador de recursos também pode limitar o número de sessões do usuário que podem estar ativas simultaneamente para cada grupo de consumidores de recursos. Outros recursos que podem ser controlados pelo gerenciador de recursos incluem o espaço de undo.

Oracle Enterprise Manager

Oracle Enterprise Manager é a principal ferramenta do Oracle para gerenciamento de sistemas de banco de dados. Ele oferece uma interface gráfica com o usuário (GUI) fácil de usar para a maioria das tarefas associadas à administração de um banco de dados Oracle, incluindo configuração, monitoração de desempenho, gerenciamento de recursos, gerenciamento de segurança e acesso a vários conselheiros.

Mineração de dados

O Oracle Data Mining oferece uma série de algoritmos que incorporam o processo de mineração de dados dentro do banco de dados tanto para criar um modelo sobre um conjunto de dados de treinamento quanto para aplicar o modelo a fim de contar os dados reais de produção. O fato de que os dados nunca precisam sair do banco de dados é uma vantagem significativa sobre o uso de um mecanismo de mineração de dados. Ter de extrair e inserir conjuntos de dados potencialmente muito grandes para um mecanismo separado é complicado e dispendioso e pode impedir que novos dados sejam contados instantaneamente enquanto são inseridos no banco de dados. O Oracle oferece funcionalidade para aprendizado supervisionado e não supervisionado, incluindo:

- Classificação
- Regressão
- Importância de atributo
- Agrupamento
- Análise de cesta de mercado
- Extração de características
- Exploração de texto
- Bioinformática (BLAST)

O Oracle oferece duas interfaces para a funcionalidade de mineração de dados: uma interface Java e outra baseada na linguagem procedural PL/SQL do Oracle. Quando um modelo tiver sido criado sobre um banco de dados Oracle, ele pode ser enviado para ser implantado em outros bancos de dados Oracle. O Oracle pode importar e exportar modelos usando uma representação que é baseada na representação PL/SQL nativa ou na Predictive Model Markup Language (PMML) padrão. Um modelo PMML gerado pelo Oracle também pode ser consumido por outras ferramentas que admitem PMML.

Notas bibliográficas

Informações de produto atualizadas, incluindo a documentação sobre produtos Oracle, podem ser encontradas nos sites `http://www.oracle.com` e `http://technet.oracle.com`.

A indexação extensível no Oracle é descrita em Srinivasan *et al.* [2000b] e Srinivasan *et al.* [2000c]. Tabelas organizadas em índice são descritas em Srinivasan *et al.* [2000a]. Banerjee *et al.* [2000], Murthy e Banerjee [2003] e Krishnaprasad *et al.* [2004] descrevem o suporte para XML no Oracle. Bello *et al.* [1998] descrevem views materializadas no Oracle. Antoshenkov [1995] descreve a técnica de compactação de mapa de bits alinhados em byte usada no Oracle; veja também Johnson [1999]. Lahiri *et al.* [2001b] descrevem a funcionalidade de fusão de cache do Oracle Real Application Clusters.

A recuperação no Oracle é descrita em Joshi *et al.* [1998] e Lahiri *et al.* [2001a]. Mensagens e filas no Oracle são descritas em Gawlick [1998]. Witkowski *et al.* [2003a] e Witkowski *et al.* [2003b] descrevem a cláusula MODEL.

Os algoritmos de gerenciamento de memória para classificação e hashing são descritos em Dageville e Zait [2002]. Poess e Potapov [2003] descrevem o recurso de compactação de tabela do Oracle. O algoritmo do conjunto de itens freqüentes para análise de cesta de mercado no Oracle Data Mining é descrito em Li e Mozes [2004].

O ajuste automático da SQL é descrito em Dageville *et al.* [2004b]. Cruanes *et al.* [2004] descrevem a execução paralela no Oracle.

CAPÍTULO 28

IBM DB2 Universal Database

Sriram Padmanabhan
IBM T. J. Watson Research Center

A família de produtos DB2 Universal Database da IBM consiste em servidores de banco de dados principais e pacotes de produtos relacionados para inteligência de negócios, integração de informações e gerenciamento de conteúdo, que são bastante reconhecidos por seus recursos abrangentes e robustos. O DB2 Universal Database Server está disponível em diversas plataformas de hardware e sistema operacional. A lista de plataformas de servidores admitidas inclui sistemas de alto nível, como mainframes, processadores maciçamente paralelos (MPP) e grandes servidores com multiprocessadores simétricos (SMP); sistemas de média escala, como SMPs de quatro e oito vias; workstations; e até mesmo pequenos dispositivos de mão. Os sistemas operacionais que são aceitos incluem variantes do Unix como Linux, AIX, Solaris e HP-UX, além de Windows 2000, Windows XP, MVS, VM, OS/400 e diversos outros. O DB2 Everyplace Edition tem suporte para sistemas operacionais como PalmOS e Windows CE. O DB2 Cloudscape é um mecanismo de banco de dados puramente Java, que pode ser embutido em servidores de aplicação e outras aplicações com facilidade. As aplicações podem migrar de modo transparente de plataformas inferiores para servidores de alto nível, devido à portabilidade das interfaces e serviços do DB2. Além do motor de banco de dados básico, a família DB2 consiste em vários outros produtos que oferecem instrumentação, administração, replicação, acesso a dados distribuídos, acesso a dados difusos, OLAP e muitos outros recursos. A Figura 28.1 descreve os diferentes produtos na família.

Visão geral

A origem do DB2 pode ser acompanhada desde o projeto System R no Almaden Research Center da IBM (então chamado do IBM San Jose Research Laboratory). O primeiro produto DB2 foi lançado em 1984 na plataforma de mainframe IBM, e este foi seguido com o passar do tempo por versões para outras plataformas. As contribuições de pesquisa da IBM melhoraram continuamente o produto DB2 em áreas como processamento de transação (logging de leitura antecipada e algoritmos de recuperação ARIES), processamento e otimização de consulta (Starburst), processamento paralelo (DB2 Parallel Edition), suporte para banco de dados ativo (restrições, triggers), técnicas de consulta avançada e warehousing, como views materializadas, agrupamento multidimensional, recursos "autonômicos" e suporte objeto-relacional (ADTs, UDFs).

Como a IBM admite uma série de plataformas de servidor e sistema operacional, o motor de banco de dados DB2 consiste em quatro tipos de base de código: (1) Linux, Unix e Windows, (2) z/OS, (3) VM e (4) OS/400. Todos esses admitem um subconjunto comum da linguagem de definição de dados, SQL e interfaces administrativas. Porém, os motores possuem recursos um tanto diferentes, devido às suas origens de plataforma. Neste capítulo, o foco está no motor DB2 Universal Database (UDB) que admite Linux, Unix e Windows. Recursos específicos de interesse em outros sistemas DB2 estão destacados nas seções apropriadas.

A versão mais recente do DB2 UDB para Linux, Unix e Windows é a 8.2, que contém diversos recursos que melhoram a escalabilidade, a disponibilidade e o poder do mecanismo do DB2. Na área de escalabilidade, dois recursos significativos são *tabelas de consulta materializadas* e *agrupamento multidimensional*. Para a disponibilidade, melhorias nas áreas de utilitários on-line e replicação serão descritas. Além disso, esta versão oferece recursos autonômicos, como *conselheiro de projeto*, e ajuste e monitoração de memória automáticos. Esses e outros recursos serão descritos em suas respectivas seções.

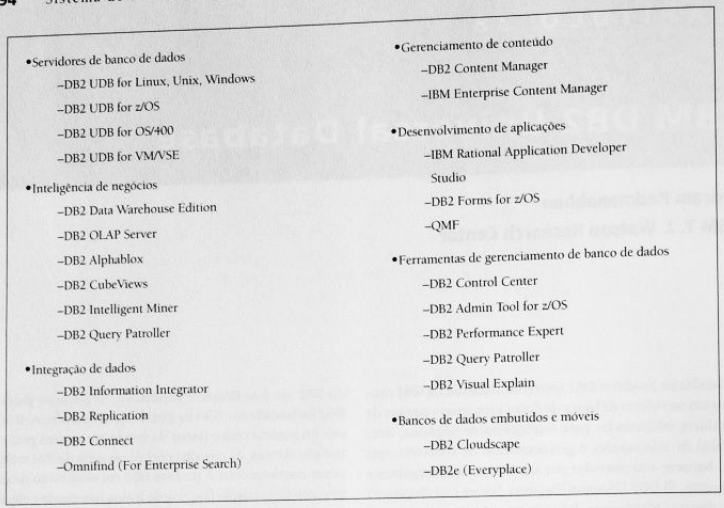


Figura 28.1 A família de produtos DB2.

Ferramentas de projeto de banco de dados

A maior parte das ferramentas de projeto de banco de dados e CASE no setor pode ser usada para o projeto de um banco de dados DB2. Em particular, ferramentas de modelagem de dados como ERWin e Rational Rose permitem que o projetista gere sintaxe DDL específica ao DB2. Por exemplo, a ferramenta UML Data Modeler do Rational Rose pode gerar instruções DDL **create distinct type** específicas do DB2 para tipos definidos pelo usuário e usá-las mais adiante nas definições de coluna. A maioria das ferramentas de projeto também admite um recurso de engenharia reversa, que lê tabelas de catálogo do DB2 e monta um projeto lógico para manipulação adicional. As ferramentas admitem a geração de restrições e índices.

O DB2 oferece suporte para muitos recursos de banco de dados lógicos e físicos usando SQL. Os recursos incluem restrições, triggers e recursão usando construções SQL. De modo semelhante, certos recursos físicos do banco de dados, como tablespaces, bufferpools e particionamento, também são admitidos pelo uso de instruções SQL. A ferramenta Control Center GUI para DB2 permite que um projetista ou administrador emita a DDL apropriada para esses recursos. Outra ferramenta, *db2look*, permite que o administrador obtenha um conjunto completo de instruções DDL para um banco de dados, incluindo tablespaces, tabelas, índices, restrições, triggers e funções que podem ser usadas para criar uma réplica exata do esquema de banco de dados para teste ou replicação.

O DB2 Control Center inclui uma série de ferramentas relacionadas a projeto e administração. Para projeto, o Control Center oferece uma visão em forma de árvore do servidor, seus bancos de dados, tabelas, views e todos os outros objetos. Ele também permite que os usuários definam novos objetos, criem consultas SQL ocasionais e vejam resultados da consulta. Ferramentas de projeto para ETL, OLAP, replicação e federação também se integram ao Control Center. A família DB2 inteira aceita o Control Center para definição de banco de dados, além das ferramentas relacionadas. O DB2 também oferece módulos de plug-in para desenvolvimento de aplicações no produto Rational Application Developer da IBM, além do produto Microsoft Visual Studio.

```
select xmlemement(name 'OC',
    xmlattributes(id_OC, datapedido),
    (select xmlagg(xmlelement(name 'item',
        xmlattributes(id_item, qtd, data_envio),
        (select xmlelement(name 'desc_item',
            xmlattributes(nome, preço))
        from produto
        where produto.id_item = item_linha.id_item)))
    from item_linha
where item_linha.id_OC = pedidos.id_OC))
from pedidos
where pedidos.id_OC= 349;
```

Figura 28.2 Consulta XML do DB2 SQL.

Variações e extensões da SQL

O DBs oferece suporte para um rico conjunto de recursos da SQL em diversos aspectos do processamento de banco de dados. Muitos dos recursos do DBs e sua sintaxe ofereceram a base para padrões na SQL-92 ou na SQL-99. Nesta seção, destacamos recursos de XML, objeto relacional e integração de aplicações no DB2 UDB versão 8.

Recursos da XML

Um rico conjunto de funções XML foi incluído no DB2. A seguir está uma lista das diversas funções XML importantes.

- **xmlelement**. Constrói uma tag de elemento com o nome indicado. Por exemplo, **xmlemement**(*livro*) cria o elemento livro.
- **xmlattributes**. Constrói o conjunto de atributos para um elemento.
- **xmlforest**. Constrói uma seqüência de elementos XML a partir de argumentos.
- **xmlconcat**. Retorna a concatenação de um número variável de argumentos XML.
- **xmlserialize**. Oferece uma versão seriada orientada a caractere do seu argumento.
- **xmlagg**. Retorna uma concatenação de um conjunto de valores XML.
- **xml2clob**. Constrói uma representação em objeto grande de caracteres (**clob**) da XML. Esse **clob** pode então ser utilizado por aplicações SQL.

As funções XML podem ser incorporadas à SQL de forma eficaz para oferecer capacidades extensivas de manipulação de XML. Por exemplo, suponha que alguém precise construir um documento XML de ordem de compra a partir das tabelas relacionais *pedidos*, *itemlinha* e *produto*, para o número de pedido 349. Na Figura 28.2, mostramos uma consulta SQL com extensões XML que podem ser usadas para criar essa ordem de compra. A saída resultante aparece na Figura 28.3.

O DB2 vem com uma capacidade de extensor de XML, que oferece aos usuários procedimentos armazenados e funções definidas pelo usuário para armazenar e manipular XML como grandes objetos de caractere ou como atributos fragmentados em tabelas. Alguém pode, então, acessar esses objetos XML por meio da SQL com as extensões XML mencionadas ou as funções extensoras da XML.

Suporte para tipos de dados

O DB2 oferece suporte para tipos de dados definidos pelo usuário (UDTs). Os usuários podem definir tipos de dados *distinct* ou *structured*. Os tipos de dados distintos são baseados nos tipos de dados embutidos do DB2. Porém, o usuário pode definir semântica adicional ou alternativa para esses novos tipos. Por exemplo, o usuário pode definir um tipo de dado distinto chamado *us_dólar*, usando

create distinct type *us_dólar* **as decimal**(9,2)

```
<OC id_OC = "349" datapedido = "2004-10-01">
  <item id_item="1", qtd="10", data_envio="2004-10-03">
  <desc_item nome = "IBM ThinkPad T41", preço = "1000.00 USD"/>
  </item>
</OC>
```

Figura 28.3 Ordem de compra em XML para id=349.

Mais tarde, o usuário pode criar um campo (por exemplo, *preço*) em uma tabela com o tipo *us_dólar*. As consultas agora podem usar o campo tipificado em predicados como o seguinte:

```
select produto from us_vendas
where preço > us_dólar(1000)
```

Tipos de dados estruturados são objetos complexos que normalmente consistem em dois ou mais atributos. Por exemplo, alguém pode usar a seguinte DDL para criar um tipo estruturado chamado *departamento_t*:

```
create type departamento_t as
        (nome_dept   varchar(32),
        chefe_dept   varchar(32),
        cont_fac     integer)
mode db2/sql

    create type ponto_t as
        (coord_x     float,
        coord_y      float)
    mode db2/sql
```

Os tipos estruturados podem ser usados para definir *tabelas tipificadas*.

```
create table dept of departamento-t
```

Pode-se criar uma hierarquia de tipos e tabelas na hierarquia que possam herdar métodos e privilégios específicos. Os tipos estruturados também podem ser usados para definir atributos aninhados dentro de uma coluna de uma tabela. Embora essa definição infrinja regras de normalização, ela pode ser adequada para aplicações orientadas a objeto, que contam com o encapsulamento e métodos bem definidos sobre objetos.

Funções e métodos definidos pelo usuário

Outro recurso importante é a capacidade para os usuários definirem suas próprias funções e métodos. Essas funções podem mais tarde ser incluídas nas instruções e consultas SQL. As funções podem gerar escalares (atributo único) ou tabelas (linha com atributos múltiplos) como seu resultado. Os usuários podem registrar funções (escalares ou tabela) usando a instrução **create function**. As funções podem ser escritas em linguagens de programação comuns, como C ou Java, ou scripts, como REXX ou Perl. Funções definidas pelo usuário (UDFs) podem operar nos modos cercado ou não cercado. No modo cercado, as funções são executadas por uma thread separada em seu próprio espaço de endereçamento. No modo não cercado, o agente de processamento de banco de dados tem permissão para executar a função no espaço de endereços do servidor. UDFs podem definir uma área de bloco de rascunho (trabalho) na qual é possível manter variáveis locais e estáticas por diferentes invocações. Assim, as UDFs podem realizar manipulações poderosas de linhas intermediárias que são suas entradas. Na Figura 28.4, mostramos uma definição de uma UDF, *db2gse.GsegeFilterDist*, em DB2, apontando para um método externo em particular, que realiza a função real.

Os métodos são outro recurso que define o comportamento dos objetos. Diferente das UDFs, eles são firmemente encapsulados com um tipo de dados estruturado em particular. Os métodos são registrados pelo uso da instrução **create method**.

Objetos grandes

Novas aplicações de banco de dados exigem a capacidade de manipular texto, imagens, vídeo e outros tipos de dados que normalmente têm um tamanho muito grande. O DB2 admite esses requisitos oferecendo três tipos de objeto grande (LOB) diferentes. Cada LOB pode ter até dois gigabytes em tamanho. Os objetos grandes em DB2 são (1) objetos grandes binários (**blobs**), (2) objetos grandes de caracteres de único byte (**clobs**) e (3) objetos grandes de caracteres de bytes duplos (**dbclobs**). O DB2 organiza esses LOBs como objetos separados, com cada linha na tabela mantendo ponteiros para seus LOBs correspondentes. Os usuários podem registrar UDFs que manipulam esses LOBs de acordo com os requisitos da aplicação.

Extensões e restrições de indexação

Um recurso recente do DB2 permite que os usuários criem extensões de índice para gerar chaves a partir de tipos de dados estruturados usando a instrução **create index extension**. Por exemplo, pode-se criar um índice sobre um atributo com base no tipo de dados *departamento_t* definido anteriormente pela geração de chaves, usando o nome do departamento. O extensor espacial do DB2 usa o método de extensão de índice para criar índices conforme mostra a Figura 28.5.

Finalmente, os usuários podem tirar proveito do rico conjunto de recursos de verificação de restrição disponíveis no DB2 para impor a semântica de objeto, como exclusividade, validade e herança.

Web Services

Na versão 8, o DB2 pode integrar Web services como produtor ou consumidor. Um Web service pode ser definido para invocar DB2, usando instruções SQL. A chamada de Web

```
create function db2gse.GsegeFilterDist (
    operation integer, g1XMin double, g1XMax double,
    g1YMin double, g1YMax double, dist double,
    g2XMin double, g2XMax double, g2YMin double,
    g2YMax double )
  returns integer
  specific db2gse.GsegeFilterDist
  external name 'db2gsefn!gsegeFilterDist'
  language C
  parameter style db2 sql
  deterministic
  not fenced
  threadsafe
  called on null input
  no sql
  no external action
  no scratchpad
  no final call
  allow parallel
  no dbinfo;
```

Figura 28.4 Definição de uma UDF.

```
create index extension db2gse.spatial-index(
        gS1 double, gS2 double, gS3 double)
from source key(geometry db2gse.ST-Geometry)
generate key using
        db2gse.GseGridIdxKeyGen(geometry..srid,
        geometry..xMin, geometry..xMax,
        geometry..yMin, geometry..yMax,
        gS1, gS2, gS3)

with target key(srsId integer,
        lvl integer, gX integer, gY integer, xMin double,
        xMax double, yMin double, yMax double)
search methods <condições> <ações>
```

Figura 28.5 Extensão de índice espacial no DB2.

service resultante é processada por um mecanismo de Web service embutido em DB2, e a resposta SOAP apropriada que é gerada. Por exemplo, se houver um Web service chamado *GetRecentActivity*(*id_cli*) que invoca a seguinte SQL, o resultado deve ser a última transação para esse cliente.

```
select id_trn, valor, data
from transações
where id_cli = <input>
order by date
fetch first 1 row only;
```

A SQL a seguir mostra o DB2 atuando como um consumidor de um Web service. Nesse exemplo, a função definida pelo usuário GetQuote( ) é um Web service. O DB2 faz a chamada do Web service usando um mecanismo de Web service embutido. Nesse caso, *GetQuote* retorna um valor de ação numérico para cada *id_ticket* na tabela de portfólio.

```
select ticker_id, GetQuote(ticker_id)
from portfolio
```

Filas de mensagens

O DB2 também admite o produto Websphere MQ da IBM definindo UDFs apropriados. Os UDFs são definidos para interfaces de leitura e escrita. Esses UDFs podem ser incorporados em SQL para leitura ou escrita para filas de mensagem.

Armazenamento e indexação

A arquitetura de armazenamento e indexação no DB2 consiste na camada do sistema de arquivos e gerenciamento de disco, serviços para gerenciar pools de buffer, objetos de dados como tabelas, LOBs, objetos de índice e gerenciadores de concorrência e recuperação. Passamos pela arquitetura de armazenamento geral nesta seção. Além disso, descrevemos um novo recurso no DB2 versão 8, chamado agrupamento multidimensional, na próxima seção.

Arquitetura de armazenamento

O DB2 oferece abstrações de armazenamento para gerenciar tabelas lógicas de banco de dados de forma útil em um ambiente de multinós e multidisco. Os *grupos de nós* podem ser definidos para dar suporte ao particionamento de tabela por um conjunto específico de nós em um sistema multinós. Isso permite a flexibilidade completa na alocação de partições de tabela para diferentes nós em um sistema. Por exemplo, grandes tabelas podem ser particionadas por todos os nós em um sistema, enquanto pequenas tabelas podem residir em um único nó.

Dentro de um nó, o DB2 usa *tablespaces* para organizar tabelas. Uma tablespace consiste em um ou mais *containers*, que são referências a diretórios, dispositivos ou arquivos. Uma tablespace pode conter zero ou mais objetos de banco de dados como tabelas, índices ou LOBs. A Figura 28.6 ilustra esses conceitos. Nessa figura, duas tablespaces foram definidas para um grupo de nós. A tablespace *rec_humanos* recebe quatro containers, enquanto a tablespace *sched* tem apenas um container. As tabelas *funcionário* e *departamento* são atribuídas à tablespace *rec_humanos*, enquanto a tabela *projeto* está na tablespace *sched*. O espalhamento é usado para alocar containers da tablespace *rec_humanos*. O DB2 permite que o administrador crie tablespaces gerenciadas pelo sistema ou gerenciadas pelo *DBMS*. Os espaços gerenciados pelo sistema (SMS) são diretórios ou sistemas de arquivos que são mantidos pelo sistema operacional básico. No SMS, o DB2 cria objetos de arquivo nos diretórios e aloca dados a cada um dos arquivos. Os espaços gerenciados por dados (DMS) são dispositivos brutos ou arquivos pré-alocados que são então controlados pelo DB2. O tamanho desses containers nunca pode aumentar ou diminuir. O DB2 cria mapas de alocação e gerencia a própria tablespace DMS. Nos dois casos, uma extensão das páginas é a unidade de gerenciamento de espaço. O administrador pode escolher o tamanho de extensão para uma tablespace.

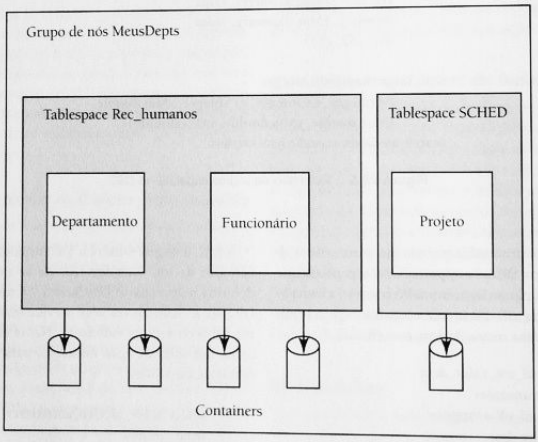


Figura 28.6 Tablespaces e containers em DB2.

O DB2 admite o espalhamento pelos diferentes containers como um comportamento padrão. Por exemplo, quando os dados são inseridos em uma tabela recém-criada, a primeira extensão é atribuída a um container. Quando a extensão está cheia, os próximos itens de dados são alocados ao próximo container em forma de rodízio. O espalhamento oferece dois benefícios significativos: E/S paralela e balanceamento de carga.

Pools de buffer

Um ou mais pools de buffer podem estar associados a cada tablespace para gerenciar diferentes objetos, como dados e índices. O pool de buffer é uma área de dados compartilhada comum, que mantém cópias dos objetos na memória. Esses objetos normalmente são organizados como páginas para gerenciamento no pool de buffers. O DB2 permite que pools de buffer sejam definidos por instruções SQL. O DB2 versão 8 tem a capacidade de crescer ou encurtar pools de buffer on-line e também automaticamente, escolhendo a configuração **automatic** para o parâmetro de configuração do pool de buffers. Um administrador pode acrescentar mais páginas a um pool de buffer ou diminuir seu tamanho sem diminuir a atividade do banco de dados.

```
create bufferpool <pool de buffers> ....
alter bufferpool <pool de buffers> size <n>
```

O DB2 também admite a pré-busca e escritas assíncronas usando threads. O componente gerenciador de dados dispara a pré-busca de dados e páginas de índice com base nos padrões de acesso de consulta. Por exemplo, uma varredura de tabela sempre dispara a pré-busca de páginas de dados. As varreduras de índice podem disparar a pré-busca de páginas de índice, além de páginas de dados, se estiverem sendo acessados em um padrão agrupado. O número de pré-buscadores e o tamanho da pré-busca são parâmetros configuráveis que precisam ser inicializados de acordo com o número de discos ou containers na tablespace.

Tabelas, registros e índices

O DB2 organiza os dados relacionais como registros em páginas. A Figura 28.7 mostra a visão lógica de uma tabela e um índice associado. A tabela consiste em um conjunto de páginas. Cada página consiste em um conjunto de registros que são registros de dados do usuário ou registros especiais do sistema. A página zero da tabela contém registros especiais do sistema sobre a tabela e seu status. O DB2 usa um registro de mapa de espaço chamado registro de controle de espaço livre (FSCR – Free Space Control Record) para encontrar o espaço livre na tabela. O registro FSCR normalmente contém um mapa de espaço para 500 páginas. A entrada do FSCR é uma máscara de bits que oferece uma indicação aproximada da possibilidade de espaço livre em uma página. O algoritmo de inserção e atualização precisa validar as entradas do FSCR realizando uma verificação física do espaço disponível em uma página.

Os índices também são organizados como páginas contendo registros de índice e ponteiros para páginas filha e irmã. O DB2 oferece suporte para os mecanismos de índice de árvore B^+ internamente. O índice de árvore B^+ contém páginas internas e páginas de folha. Os índices possuem ponteiros bidirecionais no nível de folha para dar suporte a varreduras direta e reversa. As páginas de folha contêm entradas de índice que apontam para registros na tabela. Cada registro na tabela pode ser identificado exclusivamente usando sua informação de página e slot, que é chamada *ID de registro* ou RID.

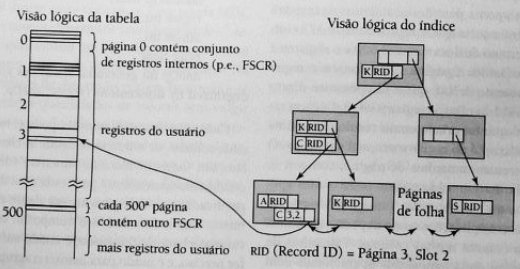


Figura 28.7 Visão lógica de tabelas e índices no DB2.

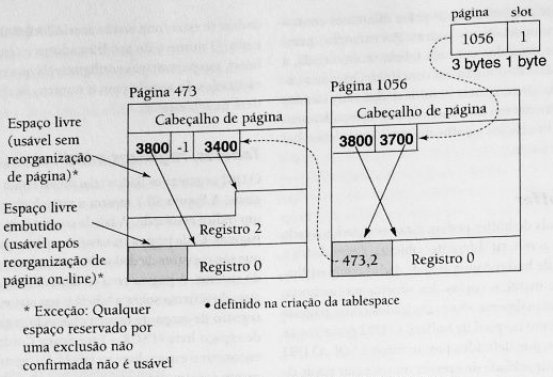


Figura 28.8 Página de dados e layout de registro em DB2.

O DB2 admite "colunas de inclusão" na definição de índice, como:

create unique index I1 **on** T1 (C1) **include** (C2)

As colunas de índice incluídas permitem que o DB2 estenda o uso de técnicas de processamento de consulta "somente de índice" sempre que possível. Diretivas adicionais como **minpctused** e **pctfree** podem ser usadas para controlar a alocação de espaço de merge e inicial das páginas de índice.

A Figura 28.8 mostra o formato típico da página de dados no DB2. Cada página de dados contém um cabeçalho e um diretório de slot. O diretório de slot é um array de 255 entradas que aponta para deslocamentos de registro na página. A figura mostra que a página número 473 contém o registro zero no deslocamento 3.800 e o registro 2 no deslocamento 3.400. A página 1.056 contém o registro 1 no deslocamento 3.700, que é um ponteiro direto para o registro <473,2>. Daí, o registro <473,2> é um registro de estouro que foi criado como resultado de uma operação de atualização do registro original <1056,1>. O DB2 admite diferentes tamanhos de página, como 4, 8, 16 e 32 kilobytes. Porém, cada página pode conter apenas 255 registros do usuário nela. Tamanhos de página maiores são úteis em aplicações como depósito de dados, em que a tabela contém muitas colunas. Tamanhos de página menores são úteis para dados operacionais com atualizações freqüentes.

Agrupamento multidimensional

Esta seção oferece uma rápida visão geral dos principais recursos do agrupamento multidimensional (MDC). Com esse recurso, uma tabela do DB2 pode ser criada especificando uma ou mais chaves como dimensões ao longo das quais os dados da tabela são agrupados. Criamos uma nova cláusula chamada **organize by dimensions** para essa finalidade. Por exemplo, a DDL a seguir descreve uma tabela vendas organizada por atributos *idLoja*, *year*(*dataPedido*) e *idItem* como dimensões.

create table *vendas*(*idLoja* **int**,
dataPedido **date**,
dataEnvio **date**,
dataReceb **date**,
region **int**,
idItem **int**,
price **float**
anoDp **int generated always as** *year*(*dataPedido*))
organized by dimensions (*região*, *anoDp*, *idItem*)

Cada uma dessas dimensões pode consistir em uma ou mais colunas, de modo semelhante às chaves de índice. De fato, um 'índice de bloco de dimensão' (descrito a seguir) é criado automaticamente para cada uma das dimensões especificadas e é usado para acessar dados rápida e eficientemente. Um índice de bloco composto, contendo todas as colunas-chave de dimensão, é criado automaticamente, se for preciso, e é usado para manter o agrupamento de dados sobre a atividade de insert e update.

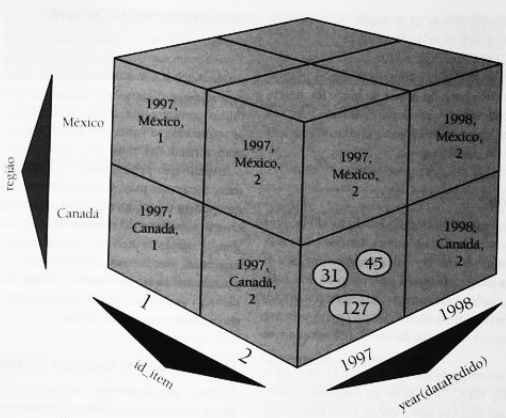


Figura 28.9 Visão lógica do layout físico de uma tabela MDC.

Cada combinação exclusiva de valores de dimensão forma uma "célula" lógica, fisicamente organizada como blocos de páginas, em que um bloco é um conjunto de páginas consecutivas no disco. O conjunto de blocos que contêm páginas com dados que possuem um certo valor de chave de um dos índices de bloco de dimensão é chamado de "fatia". Cada página da tabela faz parte de exatamente um bloco, e todos os blocos da tabela consistem no mesmo número de páginas, a saber, o tamanho do bloco. O DB2 tem associado o tamanho do bloco com o tamanho de extensão da tablespace, de modo que os limites de bloco se alinham com os limites de extensão.

A Figura 28.9 ilustra esses conceitos. Essa tabela MDC é agrupada junto com as dimensões *year*(*dataPedido*),[1] *região* e *idItem*. A figura mostra um cubo lógico simples com apenas dois valores para cada atributo de dimensão. Na realidade, os atributos de dimensão podem facilmente se estender para grandes quantidades de valores sem exigir qualquer administração. As células lógicas são representadas pelos subcubos na figura. Os registros na tabela são armazenados em blocos, que contêm o tamanho de uma extensão das páginas consecutivas no disco. No diagrama, um bloco é representado por uma oval sombreada, e é numerado de acordo com a ordem lógica das extensões alocadas na tabela. Mostramos apenas alguns blocos de dados para a célula identificada pelos valores de dimensão <1997,Canada,2>. Uma coluna ou linha na grade representa uma fatia para uma dimensão em particular. Por exemplo, todos os registros contendo o valor "Canada" na dimensão *região* são encontrados nos blocos contidos na fatia definida pela coluna "Canada" no cubo. De fato, cada bloco nessa fatia só contém registros com "Canada" no campo *região*.

1. As dimensões podem ser criadas por meio de uma função gerada.

Índices de bloco

Em nosso exemplo, um índice de bloco de dimensão é criado em cada um dos atributos *year*(*dataPedido*), *região* e *idItem*. Cada índice de bloco de dimensão é estruturado da mesma maneira que um índice de árvore B tradicional, exceto que, no nível de folha, as chaves apontam para um *identificador de bloco* (BID) em vez de um identificador de registro (RID). Como cada bloco contém potencialmente muitas páginas de registros, esses índices de bloco são muito menores do que os índices de RID e só precisam ser atualizados quando um novo bloco é adicionado a uma célula ou blocos existentes são esvaziados e removidos de uma célula. Uma fatia, ou o conjunto de blocos contendo páginas com todos os registros tendo um valor de chave em particular em uma dimensão, são representados no índice de bloco de dimensão associado por uma lista BID para esse valor de chave. A Figu-

ra 28.10 ilustra fatias de blocos para valores específicos de dimensões de *região* e *idItem*, respectivamente.

Nesse exemplo, para encontrar a fatia que contém todos os registros com "Canada" para a dimensão *região*, pesquisaríamos esse valor de chave no índice de bloco da dimensão *região* e encontraríamos uma chave conforme mostra a Figura 28.10a. Essa chave aponta para o conjunto exato de BIDs para o valor em particular.

Mapa de bloco

Um mapa de bloco também está associado à tabela. Esse mapa registra o estado de cada bloco pertencente à tabela. Um bloco pode estar em uma série de estados como **in use**, **free**, **loaded**, **requiring constraint enforcement**. Os estados do bloco são usados pela camada de gerenciamento de dados a fim de determinar várias opções de processamento. A Figura 28.11 mostra um mapa de bloco de exemplo para uma tabela.

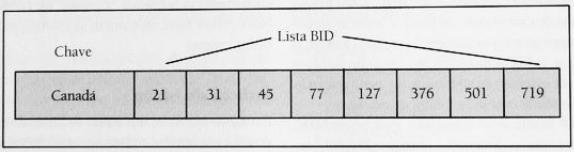
O elemento 0 no mapa de bloco representa o bloco 0 no diagrama da tabela MDC. Seu status de disponibilidade é "U", indicando que está em uso. Porém, é um bloco especial e não contém quaisquer registros do usuário. Os blocos 2, 3, 9, 10, 13, 14 e 17 não estão sendo usados na tabela e são considerados "F", ou livres, no mapa de bloco. Os blocos 7 e 18 recentemente foram carregados para a tabela. O bloco 12 foi carregado anteriormente e exige que uma verificação de restrição seja realizada sobre ele.

Considerações de projeto

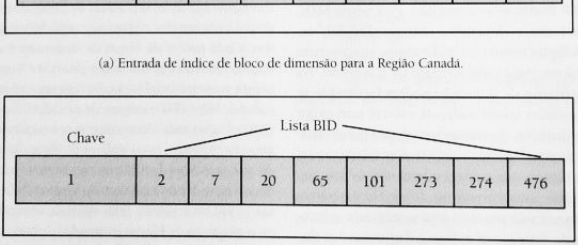
Um aspecto crucial do MDC é escolher o conjunto certo de dimensões para o agrupamento de uma tabela e o parâmetro de tamanho de bloco certo para reduzir a utilização de espaço. Se as dimensões e tamanhos de bloco forem escolhidos corretamente, então os benefícios do agrupamento se traduzem em vantagens significativas de desempenho e manutenção. Por outro lado, se escolhidos incorretamente, o desempenho pode diminuir e a utilização de espaço poderia ser muito pior. Existem diversos controles de ajuste que podem ser explorados para organizar a tabela. Estes incluem a variação do número de dimensões, a variação da granularidade de uma ou mais dimensões, a variação do tamanho do bloco (tamanho de extensão) e o tamanho de página da tablespace contendo a tabela. Uma ou mais dessas técnicas poderá ser usada em conjunto para identificar a melhor organização da tabela.

Impacto sobre técnicas existentes

É natural perguntar se o novo recurso MDC possui um impacto negativo ou se desativa alguns recursos existentes do DB2 para tabelas normais. Todos os recursos existentes, como índices RID secundários, restrições, triggers, definição de views materializadas e opções de processamento de consulta, estão disponíveis para tabelas MDC. Logo, as tabelas MDC se comportam como tabelas normais, exceto por seus aspectos melhorados de agrupamento e processamento.



(a) Entrada de índice de bloco de dimensão para a Região Canadá.



(b) Entrada de índice de bloco de dimensão para o item Id = 1.

Figura 28.10 Entradas de chave do índice de bloco.

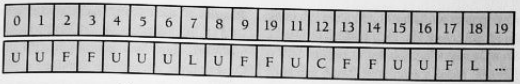


Figura 28.11 Entradas de mapa de bloco.

Processamento e otimização da consulta

Consultas DB2 são transformadas em uma árvore de operações pelo compilador de consulta. A árvore de operadores de consulta é usada no momento da execução para processamento. O DB2 admite um rico conjunto de operadores de consulta, que lhe permite considerar as melhores estratégias de processamento e oferece a flexibilidade para executar tarefas de consulta complexas.

As Figuras 28.12 e 28.13 mostram uma consulta e seu plano de consulta associado no DB2. A consulta é uma consulta complexa representativa (consulta 5) do benchmark TPC-H e contém várias junções e agregações. O plano de consulta escolhido para esse exemplo em particular é bem simples, pois muitos índices e outras estruturas auxiliares, como views materializadas, não foram definidos para essas tabelas. O DB2 oferece diversas facilidades de "explicação", incluindo um poderoso recurso visual de explicação no Control Center, que pode ajudar os usuários a entender os detalhes de um plano de execução de consulta. O plano de consulta mostrado na figura é baseado na explicação visual para a consulta. A explicação visual permite que o usuário entenda o custo e outras propriedades relevantes das diferentes operações do plano de consulta.

Todas as consultas e instruções SQL, por mais complexas que possam ser, são transformadas em uma árvore de consulta. Os operadores de base ou folha da árvore de consulta manipulam registros em tabelas de banco de dados. Essas operações também são chamadas *métodos de acesso*. Operações intermediárias da árvore incluem operações de álgebra relacional como junção, operações de conjunto e agregação. A raiz da árvore produz os resultados da consulta ou instrução SQL.

Métodos de acesso

O DB2 admite um conjunto abrangente de métodos de acesso sobre tabelas relacionais. A lista de métodos de acesso inclui:

- **Varredura de tabela.** Esse é o método mais básico e realiza um acesso página a página de todos os registros na tabela.
- **Varredura de índice.** Um índice é usado para selecionar os registros específicos que satisfazem a consulta. Os registros de qualificação são acessados por meio de RIDs no índice. O DB2 detecta oportunidades para a prébusca de páginas de dados quando observa um padrão de acesso sequencial.

```
— — 'TPCD Local Supplier Volume Query (Q5)';
select n_nome, sum(l_preçoestendido*(1-l_desconto)) as receita
from tpcd.cliente, tpcd.pedidos, tpcd.itemlinha,
    tpcd.fornecedor, tpcd.nação, tpcd.região
where c_chavecli = o-custkey and
    o_chavepedido = l-orderkey and
    l_chaveforn = s_chaveforn and
    c_chavenação = s_chavenação and
    s_chavenação = n_chavenação and
    n_chaveregião = r_chaveregião and
    r_nome = 'MIDDLE EAST' and
    o_datapedido >= date('1995-01-01') and
    o_datapedido < date('1995-01-01') + 1 year
group by n_nome
order by receita desc;
```

Figura 28.12 Consulta SQL.

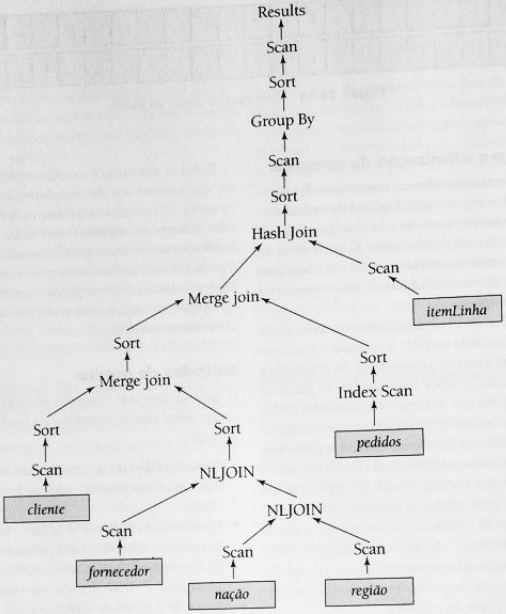


Figura 28.13 Plano de consulta do DB2 (explicação gráfica).

- **Varredura de índice de bloco**. Esse é um novo método de acesso para tabelas MDC. Um dos índices de bloco é usado para varrer um conjunto específico de blocos de dados MDC. Os blocos de qualificação são acessados e processados nas operações de varredura da tabela de bloco.
- **Apenas índice**. Nesse caso, o índice contém todos os atributos que são exigidos pela consulta. Logo, uma varredura das entradas de índice é suficiente. A técnica somente de índice normalmente é uma boa solução de desempenho.
- **Pré-busca de lista**. Esse método de acesso é escolhido para uma varredura de índice não agrupado com um número significativo de RIDs. O DB2 possui uma operação sort sobre os RIDs e realiza uma busca dos registros em ordem classificada a partir das páginas de dados. O acesso classificado muda o padrão de E/S de aleatório para sequencial, e também permite oportunidades de pré-busca. A pré-busca de lista foi estendida para lidar também com índices de bloco.
- **AND de índice de bloco e registro**. Esse método é usado quando o DB2 determina que mais de um índice pode ser usado para restringir o número de registros satisfatórios em uma tabela básica. O índice mais seletivo é processado para gerar uma lista de BIDs ou RIDs. O próximo índice seletivo é então processado para retornar os BIDs ou RIDs que ele qualifica. Um BID ou RID só se qualifica para mais processamento se estiver presente na interseção (operação AND) dos resultados de varredura de índice. O resultado de uma operação AND de índice é uma pequena lista de BIDs ou RIDs qualificados que são usados para buscar os registros correspondentes na tabela básica.
- **Ordenação de índice de bloco e registro**. Essa estratégia é usada se dois ou mais índices de bloco ou registro pu-

derem ser usados para satisfazer predicados de consulta que são combinados pelo uso do operador OR. O DB2 elimina BIDs ou RIDs duplicados realizando um sort e depois buscando o conjunto de registros resultante. O OR de índice foi estendido para considerar combinações de índice de bloco e RID.

Todos os predicados de seleção e projeção de uma consulta normalmente são "empurrados para baixo" para os métodos de acesso. Além disso, o DB2 realiza certas operações como classificação e agregação no modo "push-down" a fim de reduzir os caminhos de instrução.

Esse recurso de MDC tira proveito do novo conjunto de melhorias de método de acesso para varreduras de índice de bloco, pré-busca de índice de bloco, AND de índice de bloco e OR de índice de bloco para processar blocos de dados.

Operações de junção, agregação e conjunto

O DB2 admite uma série de técnicas para essas operações. Para junção, o DB2 pode escolher entre as técnicas de loop aninhado, sort-merge e junção de hash. Ao descrever as operações binárias de junção e conjunto, usamos a notação de tabelas "externas" e "internas" para distinguir os dois fluxos de entrada. A técnica de loop aninhado é útil se a tabela interna for muito pequena ou puder ser acessada usando um índice sobre um predicado de junção. Técnicas de junção sort-merge e junção de hash são usadas para junções envolvendo grandes tabelas externas e internas. Operações de conjunto são implementadas pelo uso de técnicas de sort e merge. A técnica de merge elimina duplicatas no caso de união; já no caso de interseção, as duplicatas são mantidas. O DB2 também admite operações de junção externa de todos os tipos.

O DB2 processa operações de agregação no modo antecipado ou "push-down" sempre que possível. Por exemplo, uma agregação group by pode ser realizada incorporando-se a agregação na fase de classificação. Os algoritmos de junção e agregação podem tirar proveito do processamento superescalar nas CPUs modernas, usando técnicas orientadas a bloco e cientes do cache.

Suporte para processamento SQL complexo

Um dos aspectos mais importantes do DB2 é que ele usa a infra-estrutura de processamento de consulta em um padrão extensível para dar suporte a operações SQL complexas. As operações SQL complexas incluem suporte para consultas profundamente aninhadas e correlacionadas, além de restrições, integridade referencial e triggers. Como a maioria dessas ações está embutida no plano de consulta, o DB2 é capaz de escalar e oferecer suporte para um número maior dessas restrições e ações. As restrições e verificações de integridade são montadas como operações de árvore de consulta sobre instruções SQL insert, delete ou update. O DB2 também admite manutenção de view materializada usando triggers internas.

Recursos de processamento de consulta em múltiplos processadores

O DB2 estende o conjunto básico de operações de consulta com primitivas de controle e troca de dados para dar suporte aos modos de cluster SMP, MPP e SMP de processamento de consulta. O DB2 usa uma abstração "tablequeue" para a troca de dados entre threads em diferentes nós ou no mesmo nó. A tablequeue é usada como um buffer que redireciona dados para receptores apropriados usando broadcast, multicast um a um ou direcionado. Operações de controle são usadas para criar threads e coordenar a operação de diferentes processos e threads.

Em todos esses modos, o DB2 emprega um processo coordenador para controlar as operações de consulta e a coleta do resultado final. Processos coordenadores também podem realizar algumas ações globais de processamento de banco de dados, se for preciso. Um exemplo é a operação de agregação global para combinar os resultados da agregação local. Subagentes ou threads escravas realizam as operações básicas do banco de dados em um ou mais nós. No modo SMP, os subagentes utilizam a memória compartilhada para sincronização entre si quando compartilham dados. Em um MPP, os mecanismos de tablequeue oferecem buffers e controle de fluxo para sincronizar entre diferentes nós durante a execução. O DB2 emprega técnicas extensivas para otimizar e processar consultas de forma eficiente em um ambiente MPP ou SMP. A Figura 28.14 mostra uma consulta simples executando em um sistema MPP de 4 nós. Neste exemplo, a tabela *vendas* é particionada pelos quatro nós $P_1, \ldots, P_4$. A consulta é executada criando-se agentes que são executados em cada um desses nós para varrer e filtrar as linhas da tabela vendas nesse nó (chamado envio de função) e as linhas resultantes são enviadas ao nó coordenador.

Otimização de consulta

O compilador de consulta do DB2 utiliza uma representação interna da consulta, chamada modelo gráfico de consulta (QGM – Query-Graph Model), a fim de realizar transformações e otimizações. Depois de analisar a instrução SQL, o DB2 realiza transformações semânticas sobre o QGM para impor restrições, integridade referencial e trig-

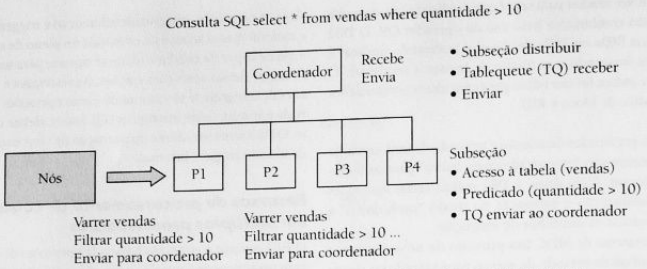


Figura 28.14 Processamento de consulta MPP do DB2 usando o envio de função.

gers. O resultado dessas transformações é um QGM melhorado. Em seguida, o DB2 tenta realizar transformações de *reescrita de consulta* que são consideradas benéficas em sua maior parte. As regras de reescrita são disparadas se forem aplicáveis para realizar as transformações exigidas. Alguns exemplos de transformações de reescrita incluem (1) descorrelacionando subconsultas correlacionadas, (2) transformando certas subconsultas em junções usando o processamento antecipado, (3) empurrando a operação **group by** para baixo das junções, se for o caso, e (4) usando views materializadas para partes da consulta original.

O componente otimizador de consulta usa esse QGM melhorado e transformado como entrada para otimização. O otimizador é baseado em custo e usa uma estrutura extensível, controlada por regras. O otimizador pode ser configurado para operar em diferentes níveis de complexidade. No nível mais alto, ele usa um algoritmo de programação dinâmico para considerar todas as opções de plano de consulta e escolhe o plano de custo ideal. Em um nível intermediário, o otimizador não considera certos planos, métodos de acesso (por exemplo, OR de índice) ou regras de reescrita. No nível de complexidade mais baixo, o otimizador usa uma heurística gulosa simples para escolher um plano de consulta bom, mas não necessariamente ideal. O otimizador usa modelos detalhados das operações de processamento de consulta, incluindo tamanhos de memória e pré-busca, para obter estimativas precisas dos custos de E/S e CPU. Ele conta com as estatísticas dos dados para estimar a cardinalidade e as seletividades das operações. O DB2 permite que o usuário obtenha histogramas detalhados das distribuições em nível de coluna e combinações de colunas usando o utilitário runstats. Os histogramas detalhados contêm informações sobre as ocorrências de valor mais freqüente, além de distribuições de freqüência baseadas em quantil dos atributos. O otimizador gera um plano de consulta que é considerado o melhor plano de consulta para o nível de otimização em particular. Esse plano de consulta é convertido para threads de operadores de consulta e estruturas de dados associadas para execução pelo mecanismo de processamento de consulta.

Tabelas de consulta materializadas

Views materializadas são aceitas no DB2 versão 8 no Linux, Unix e Windows, além das plataformas z/OS. Uma view materializada pode ser qualquer definição de view geral sobre uma ou mais tabelas ou views. Uma view materializada é útil por manter uma cópia persistente dos dados de view e permitir o processamento de consulta mais rápido. No DB2, essas views materializadas são chamadas tabelas de consulta materializadas (MQTs – Materialized Query Tables). MQTs são especificadas com o uso da instrução **create table**, como mostramos com o exemplo da Figura 28.15.

No DB2, as MQTs podem referenciar outras MQTs para criar uma árvore ou floresta de views dependentes. Essas MQTs são altamente escaláveis, pois podem ser particionadas em um ambiente MPP e podem ter chaves de agrupamento MDC. As MQTs são mais valiosas se o mecanismo de banco de dados puder rotear consultas para elas de forma transparente e também se o mecanismo de banco de dados puder mantê-las eficientemente sempre que possível. O DB2 oferece esses dois recursos.

Roteamento de consulta para MQTs

A infra-estrutura do compilador de consulta no DB2 é adequada de forma ideal para aproveitar todo o poder das MQTs. O modelo QGM interno permite que o compilador combine a consulta de entrada com as definições de MQT disponíveis e escolha MQTs apropriadas para consideração. Depois de combinar, o compilador considera várias

```
create table dept_func(id_dept integer, id_func integer,
        nome_func varchar(100), id_ger integer) as
    select id_dept, id_func, nome_func, id_ger
    from funcionario, departamento
        data initially deferred
        refresh immediate ---- (ou deferred)
        maintained by user ---- (ou system)
```

Figura 28.15 Tabelas de consulta materializadas do DB2.

opções para otimização. Entre elas estão a consulta básica e também versões de redirecionamento de MQT adequadas. O otimizador percorre essas opções antes de escolher a versão ideal para execução. O fluxo inteiro do redirecionamento e otimização aparece na Figura 28.16.

Manutenção de MQTs

MQTs são úteis somente se o mecanismo de banco de dados oferecer técnicas eficientes para manutenção. Existem duas dimensões para a manutenção: tempo e custo. Na dimensão tempo, as duas opções são *immediate* (imediato) e *deferred* (adiado). O DB2 aceita essas duas opções. Se alguém selecionar imediato, então triggers internas são criadas e compiladas para as instruções insert, update ou delete dos objetos de origem a fim de processar as atualizações para as MQTs dependentes. No caso da manutenção adiada, as tabelas atualizadas são movidas para um modo de integridade e uma instrução **refresh** explícita precisa ser emitida de modo a realizar a manutenção. Na dimensão tamanho, as opções são *incremental* ou *total*. A manutenção incremental implica que somente as linhas atualizadas recentemente devem ser usadas para manutenção. A manutenção completa implica que a MQT inteira será atualizada a partir de suas origens. A matriz na Figura 28.17 mostra as duas dimensões e as opções que são mais úteis ao longo dessas dimensões. Por exemplo, a manutenção imediata e completa não são compatíveis, a menos que as origens sejam extremamente pequenas. O DB2 também considera que as MQTs serão mantidas pelo **usuário**. Nesse caso, o refresh das MQTs é determinado pelos usuários realizando o processamento explícito por meio da SQL ou de utilitários.

Os comandos a seguir oferecem um exemplo simples da realização da manutenção adiada para a view materializada *dept_func* após uma operação de carga para uma de suas origens.

```
load from novosdados.txt of type del
insert into funcionário;

refresh table dept_func
```

Recursos autonômicos no DB2

O DB2 UDB versão 8.2 oferece recursos para simplificar o projeto e o manuseio dos bancos de dados. A computação autonômica compreende um conjunto de técnicas que permitem que o ambiente de computação seja autogerenciado e reduza as dependências externas diante de mudanças externas e internas na segurança, carga do sistema e outros fatores. Configuração, otimização, proteção e monitoração são exemplos de áreas que se beneficiam com melhorias de computação autonômica. As próximas seções descrevem rapidamente as áreas de configuração e otimização.

Configuração

O DB2 está oferecendo suporte para o ajuste automático de diversos parâmetros de configuração de memória e sistema. Por exemplo, parâmetros como tamanhos de pool de buffer e tamanhos de heap de classificação podem ser especificados como automáticos. Nesse caso, o DB2 monitora o sistema e lentamente aumenta ou diminui esses tamanhos de memória heap, dependendo das características da carga de trabalho.

Otimização

Estruturas de dados auxiliares (índices, MQTs) e recursos de organização de dados (particionamento, agrupamento) são aspectos importantes da melhoria do desempenho do processamento de banco de dados no DB2. No passado, o administrador de banco de dados (DBA) tinha de usar experiências e orientações conhecidas para escolher índices significativos, MQTs, chaves de partição e chaves de agrupamento. Dado o número de opções em potencial, até mesmo os melhores especialistas não são capazes de encontrar a combinação certa desses recursos para determinada carga de trabalho em pouco tempo. O DB2 versão 8.2 apresenta o *Design Advisor*, que oferece conselhos baseados em carga de trabalho para todos esses recursos. A ferramenta Design Advisor analisa automaticamente uma carga de trabalho,

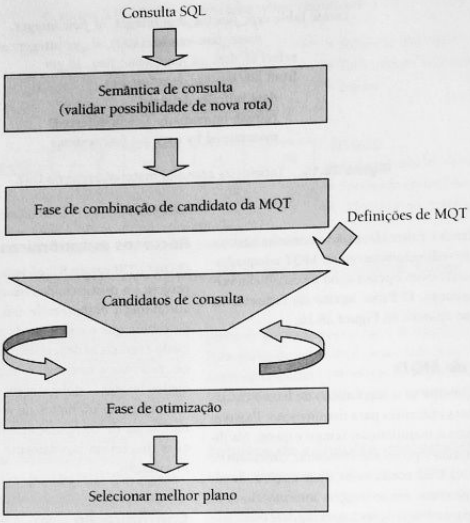


Figura 28.16 Combinação e otimização de MQT no DB2.

Opções	*Incremental*	*Total*
Immediate	Sim, após insert/update/delete	Normalmente não
Deferred	Sim, após a carga	Sim

Figura 28.17 Opções para manutenção de MQT no DB2.

usando técnicas de otimização para apresentar um conjunto de recomendações. A sintaxe de comando do Design Advisor é:

```
db2advis –d <nome BD> –i <arqcargatrabalho> –m MICP
```

O parâmetro "-m" permite que o usuário especifique as seguintes opções:

- M – Tabelas de consulta materializadas
- I – Índices
- C – Agrupamento, a saber, MDC
- P – Seleção de chave de particionamento

O advisor utiliza todo o poder da estrutura de otimização de consulta do DB2 nessas recomendações. Ele usa uma carga de trabalho de entrada e restrições sobre tamanho e tempo do conselho como seus parâmetros. Como ele aproveita a estrutura de otimização do DB2, possui total conhecimento do esquema e estatísticas dos dados básicos. O advisor utiliza várias técnicas combinatórias para identificar índices, MQTs, MDCs e chaves de particionamento de modo a melhorar o desempenho de determinada carga de trabalho.

Outro aspecto da otimização é o balanceamento da carga de processamento no sistema. Em particular, os utilitários costumam aumentar a carga em um sistema e causar redução significativa no desempenho da carga de trabalho do usuário. Dada a tendência em direção a utilitários on-line,

existe uma necessidade de balancear o consumo de carga dos utilitários. O DB2 versão 8.2 introduz um mecanismo de aceleração de carga utilitário. A técnica de aceleração é baseada na teoria do controle de feedback. Ela ajusta continuamente e acelera o desempenho do utilitário de backup, usando parâmetros de controle específicos.

Ferramentas e utilitários

O DB2 oferece uma série de ferramentas para facilitar o uso e a administração. Esse conjunto básico de ferramentas é aumentado e melhorado por uma grande quantidade de ferramentas de fornecedores.

O Control Center do DB2 é a ferramenta principal para uso e administração dos bancos de dados DB2. O Control Center executa em muitas plataformas workstation. Ele é organizado a partir de objetos de dados como servidores, bancos de dados, tabelas e índices, e contém interfaces orientadas a tarefa para realizar comandos e permitir que os usuários gerem scripts SQL. A Figura 28.18 mostra uma tela do painel principal do Control Center. Essa tela mostra uma lista de tabelas no banco de dados *Sample* na instância *DB2* do nó *Crankarm*. O administrador pode usar o menu para invocar um pacote de ferramentas componentes. Os principais componentes do Control Center incluem centro de comando, centro de script, journal, gerenciamento de licença, centro de alerta, monitor de desempenho, explicação visual, gerenciamento de banco de dados remoto, gerenciamento de armazenamento e suporte para replicação. O centro de comando permite que usuários e administradores emitam comandos de banco de dados e SQL. O centro de script permite que usuários executem scripts SQL construídos interativamente ou a partir de um arquivo. O monitor de desempenho permite que usuários monitorem diversos eventos no sistema de banco de dados e obtenham snapshots do desempenho. "SmartGuides" oferecem ajuda na configuração de parâmetros e no ajuste do sistema DB2. Um criador de procedimento armazenado (stored procedure) ajuda o usuário a desenvolver e instalar procedimentos armazenados. A explicação visual permite que o usuário obtenha visões gráficas do plano de execução da consulta. Um assistente de índice ajuda o administrador sugerindo índices para melhorar o desempenho.

Embora o Control Center seja uma interface integrada para muitas das tarefas, o DB2 também oferece acesso direto à maioria das ferramentas. Para os usuários, ferramentas como a facilidade de explicação, tabelas de explicação e explicação gráfica oferecem um desmembramento detalhado dos planos de consulta. Os usuários também podem modificar estatísticas (se tiverem permissão) a fim de gerar os melhores planos de consulta.

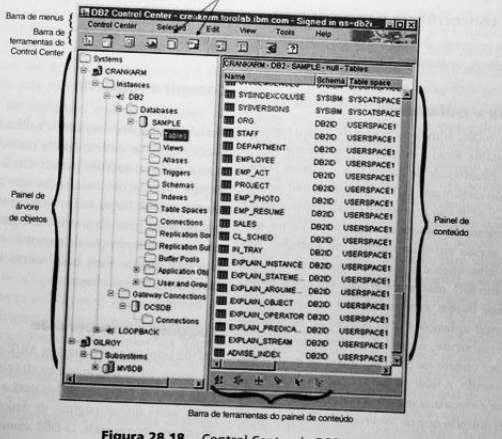


Figura 28.18 Control Center do DB2.

Utilitários

Os administradores têm o apoio de uma série de ferramentas. O DB2 oferece suporte abrangente para load, import, export, reorg, redistribute e outros utilitários relacionados a dados. Na versão 8, a maioria destes admite capacidade de processamento incremental e on-line. Por exemplo, é possível emitir um comando load no modo on-line para permitir que as aplicações acessem o conteúdo original de uma tabela simultaneamente. Os utilitários do DB2 estão todos totalmente preparados para trabalhar em modo paralelo.

Além do mais, o DB2 admite uma série de ferramentas como:

- Facilidade audit, para manter o rastro de auditoria das ações sobre o banco de dados
- Facilidade governor, para controlar a prioridade e os tempos de execução de diferentes aplicações
- Facilidade query patroller, para gerenciar as tarefas de consulta no sistema
- Facilidades trace e diagnostic, para depuração
- Facilidades de monitoração de evento, para rastrear os recursos e os eventos durante a execução do sistema

O DB2 para OS/390 possui um conjunto de ferramentas muito rico. QMF é uma ferramenta bastante usada para gerar consultas ocasionais e integrá-las às aplicações.

Controle de concorrência e recuperação

O DB2 aceita um conjunto abrangente de técnicas de controle de concorrência, isolamento e recuperação.

Concorrência e isolamento

Para isolamento, o DB2 admite os modos de *leitura repetitiva* (RR – Repeatable Read), *estabilidade de leitura* (RS – Read Stability), *estabilidade de cursor* (CS – Cursor Stability) e *leitura não confirmada* (UR – Uncommitted Read). Os modos RR, CS e UR não precisam de mais explicação. O modo de isolamento RS bloqueia somente as linhas que uma aplicação acessa em uma unidade de trabalho. Em uma varredura subsequente, a aplicação garante ver todas essas linhas (como RR), mas também pode ver novas linhas que se qualificam. Porém, essa poderia ser uma troca aceitável para algumas aplicações com relação ao isolamento RR estrito. Normalmente, o nível de isolamento padrão é CS. As aplicações podem escolher seu nível de isolamento no estágio de vínculo. A maioria das aplicações disponíveis comercialmente é vinculada usando a maioria dos níveis de isolamento, permitindo que os usuários escolham a versão correta da aplicação para o seu requisito.

Os diversos modos de isolamento são implementados pelo uso de bloqueios. O DB2 aceita bloqueios em nível de registro e de tabela. Uma estrutura de dados de tabela de bloqueio separada é mantida com as informações de bloqueio. O DB2 passa de bloqueio em nível de registro para bloqueio em nível de tabela se o espaço na tabela de bloqueio ficar pequeno. O DBs implementa o bloqueio estrito em duas fases para todas as transações de atualização. Os bloqueios de escrita ou atualização são mantidos até o momento do commit ou rollback. A Figura 28.19 mostra os diferentes modos de bloqueio e sua descrição. O conjunto de modos de bloqueio admitidos inclui bloqueios de intenção no nível de tabela, a fim de maximizar a concorrência. Além disso, o DB2 implementa o bloqueio da próxima chave e esquemas variantes para atualizações que afetam varreduras de índice, a fim de eliminar os problemas de Halloween e leitura fantasma.

A transação pode definir a granularidade do bloqueio para o nível de tabela usando a instrução **lock table**. Isso é útil para aplicações que sabem que seu nível de isolamento desejado está no nível de tabela. Além disso, o DB2 escolhe as granularidades de bloqueio apropriadas para utilitários como reorg e load. As versões offline desses utilitários normalmente bloqueiam a tabela no modo exclusivo. As versões on-line dos utilitários permitem que outras transações prossigam simultaneamente adquirindo bloqueios de linha.

Um agente de detecção de impasse é ativado para cada banco de dados e periodicamente verifica impasses entre transações. O intervalo para detecção de impasse é um parâmetro configurável. No caso de um impasse, o agente escolhe uma vítima e a aborta com um código de erro SQL de impasse.

Commit e rollback

As aplicações podem confirmar ou reverter transações usando instruções **commit** ou **rollback** explícitas. As aplicações também podem emitir instruções **begin transaction** e **end transaction** para controlar o escopo das transações. Transações aninhadas não são aceitas. Normalmente, o DB2 libera todos os bloqueios que ele mantém em favor de uma transação no **commit** ou **rollback**. Porém, se uma instrução de cursor tiver sido declarada pelo uso da cláusula **with hold**, então alguns bloqueios são mantidos entre os commits.

Logging e recuperação

O DB2 implementa o logging ARIES estrito e esquemas de recuperação. O logging de escrita antecipada é empregado para levar os registros de log para o arquivo de log persistente antes que as páginas de dados sejam escritas ou no momento do **commit**. O DB2 admite dois tipos de modos

Modo de bloqueio	Objetos	Interpretação
IN (intent none)	Tablespaces, tabelas	Leitura sem bloqueios de linha
IS (intent share)	Tablespaces, tabelas	Leitura com bloqueios de linha
NS (next key share)	Linhas	Bloqueios de leitura para níveis de isolamento RS ou CS
S (share)	Linhas, tabelas	Bloqueio de leitura
IX (intent exclusive)	Tablespaces, tabelas	Pretende atualizar linhas
SIX (share with intent exclusive)	Tabelas	Nenhum bloqueio de leitura sobre as linhas, mas bloqueios X sobre linhas atualizadas
U (update)	Linhas, tabelas	Bloqueio de atualização, mas permite que outros leiam
NX (next-key exclusive)	Linhas	Bloqueio da próxima chave para inserts/deletes, para impedir leitura fantasma durante varreduras de índice RR Permite apenas leitores não confirmados
X (exclusive)	Linhas, tabelas	Acesso exclusivo completo
Z (superexclusive)	Tablespaces, tabelas	

Figura 28.19 Modos de bloqueio do DB2.

de log: logging circular e logging de arquivamento. No logging circular, um conjunto predefinido de arquivos de log primários e secundários é utilizado. O logging circular é útil para recuperação de falhas ou recuperação de falha da aplicação. No logging de arquivamento, o DB2 cria novos arquivos de log, e os arquivos antigos precisam ser arquivados para liberar espaço no sistema de arquivos. O logging de arquivamento precisa realizar a recuperação roll-forward. Nos dois casos, o DB2 permite que o usuário configure o número de arquivos de log e os tamanhos dos arquivos de log.

Em ambientes com uso intenso da atualização, o DB2 pode ser configurado para procurar commits em grupo, a fim de reunir as escritas de log.

O DB2 admite rollback de transação e recuperação de falhas, além de recuperação de um ponto no tempo e roll-forward. No caso da recuperação de falhas, o DB2 realiza as fases-padrão de processamento de *undo* e *redo* até e a partir do último ponto de verificação, a fim de recuperar o estado confirmado apropriado do banco de dados. Para a recuperação de um ponto no tempo, o banco de dados pode ser restaurado a partir de um backup e pode ser revertido para um ponto específico no tempo, usando logs arquivados. O comando de recuperação roll-forward aceita os níveis de banco de dados e tablespace. Ele também pode ser emitido em nós específicos em um sistema de múltiplos nós. Recentemente, foi implementado um sistema de recuperação paralela, para melhorar o desempenho em sistemas SMP, utilizando muitas CPUs. O DB2 realiza a recuperação coordenada por nós MPP implementando um esquema de ponto de verificação global.

Arquitetura do sistema

A Figura 28.20 mostra alguns dos diferentes processos ou threads em um servidor DB2. As aplicações de cliente remoto se conectam ao servidor de banco de dados usando agentes de comunicação como *db2tcpcm*. Cada aplicação recebe um agente (agente coordenador nos ambientes MPP ou SMP) chamado thread *db2agent*. Esse agente e seus agentes subordinados realizam as tarefas relacionadas à aplicação. Cada banco de dados possui um conjunto de processos ou threads que realizam tarefas como busca prévia, limpeza de página do pool de buffers, logging e detecção de impasse. Finalmente, há um conjunto de agentes no nível do servidor para realizar tarefas como detecção de falha, servidor de licença, criação de processos e controle de recursos do sistema. O DB2 oferece parâmetros de configuração para controlar o número de threads e processos em um servidor. Quase todos os diferentes tipos de agentes podem ser controlados pelo uso dos parâmetros de configuração.

A Figura 28.21 mostra os diferentes tipos de segmentos de memória no DB2. A memória privada nos agentes ou threads é usada principalmente para variáveis locais e estruturas de dados que são relevantes apenas para a atividade atual. Por exemplo, um sort privado poderia alocar memória da heap privada do agente. A memória compartilhada é particionada em *memória compartilhada do servidor*, *memória compartilhada do banco de dados* e *memória compartilhada da aplicação*. A memória compartilhada no nível de banco de dados contém estruturas de dados úteis, como pool do buffer, listas de bloqueio, caches de pacote de aplicação e áreas de sort compartilhadas. As áreas do servidor e de memória compartilhada da aplicação são usadas princi-

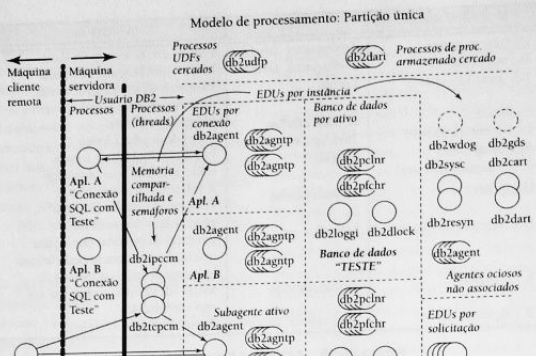


Figura 28.20 Modelo de processo no DB2.

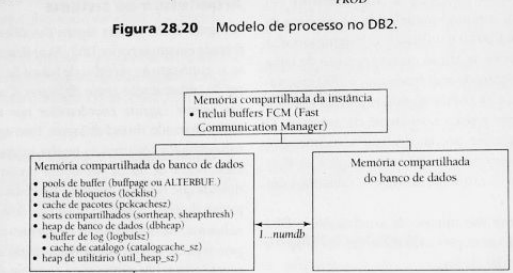


Figura 28.21 Modelo de memória do DB2.

palmente para estruturas de dados comuns e buffers de comunicação.

O DB2 admite múltiplos pools de buffers para um banco de dados. Os pools de buffers podem ser criados por meio da instrução **create bufferpool** e podem ser associados a tablespaces. Pools de buffers múltiplos são úteis por diversos motivos, mas devem ser definidos após uma análise cuidadosa dos requisitos de carga de trabalho. O DB2 aceita uma lista abrangente de parâmetros de configuração de memória e ajuste. Isso inclui parâmetros para todas as áreas grandes de heap da estrutura de dados, como o pool de buffers-padrão, a heap de sort, cache de pacotes, heaps de controle da aplicação e a área da lista de bloqueio.

Replicação, distribuição e dados externos

DB2 Replication é um produto da família DB2 que oferece capacidades de replicação entre DB2, outras origens de dados relacionais, como Oracle, Microsoft SQL Server, Sybase SQL Server e Informix, além de fontes de dados não relacionais, como IMS da IBM. Ele consiste nos componentes *capture* e *apply*, que são controlados por interfaces de administração. Os mecanismos de captura de mudança são "baseados em log" para tabelas DB2 ou "baseados em trigger" no caso de outras origens de dados. As mudanças capturadas são armazenadas em áreas de tabela de encenação temporárias, sob o controle do DB2 Replication. Essas tabelas intermediárias encenadas com as mudanças são então aplicadas às tabelas de destino, usando instruções SQL normais: inserts, updates e deletes. As transformações baseadas em SQL podem ser realizadas sobre as tabelas de encenação intermediárias usando condições de filtro e também agregações. As linhas resultantes podem ser aplicadas a uma ou mais tabelas de destino. Todas essas ações são controladas pela ferramenta de administração.

Na versão 8.2, o DB2 admite um novo recurso chamado *replicação de fila*. A replicação de fila (Q – Queue) cria um mecanismo de transporte de fila usando o produto de fila de mensagem da IBM para enviar os registros de log capturados como mensagens. Essas mensagens são extraídas das filas na extremidade receptora e aplicadas contra os destinos. O processo de aplicação pode ser feito em paralelo e leva em conta as regras de solução de conflito especificadas pelo usuário.

Outro membro da família DB2 é o produto integrador de informações do DB2, que oferece capacidades de federação, replicação (usando o mecanismo de replicação descrito anteriormente) e busca. A edição federada integra tabelas no DB2 remoto, ou outros bancos de dados relacionais, a um único banco de dados distribuído. Os usuários e desenvolvedores podem acessar diversas origens de dados não relacionais em formato tabular, usando tecnologia de wrapper. O mecanismo de federação oferece um método baseado em custo para otimização de consulta pelos diferentes sites de dados.

O DB2 admite funções de tabela definidas pelo usuário, que permitem o acesso a fontes de dados não relacionais e externas. As funções de tabela definida pelo usuário são criadas pelo uso da instrução **create function** com a cláusula **returns table**. Usando esses recursos, o DB2 é capaz de participar dos protocolos OLE DB.

Finalmente, o DB2 oferece suporte total para o processamento distribuído de transações, usando o protocolo commit de duas fases. O DB2 pode atuar como coordenador ou agente para o suporte XA distribuído. Como coordenador, o DB2 pode realizar todos os estágios do protocolo commit de duas fases. Como participante, o DB2 pode interagir com qualquer gerenciador de transações distribuído comercial.

Recursos de inteligência de negócios

DB2 Data Warehouse Edition faz parte da família DB2 que incorpora recursos de inteligência de negócios. O Data Warehouse Edition possui como alicerce o mecanismo do DB2, e o melhora com recursos para ETL, OLAP, exploração e relatório on-line. O mecanismo DB2 oferece expansão usando seus recursos de MPP. No modo MPP, o DB2 pode admitir configurações que podem se expandir para várias centenas de nós para grandes bancos de dados (terabytes). Além disso, recursos como MDC e MQT oferecem suporte para requisitos complexos de processamento de consulta da inteligência de negócios.

Outro aspecto da inteligência de negócios é o processamento analítico on-line ou OLAP. A família DB2 inclui um recurso chamado *visões de cubo* (*cube views*), que oferece um mecanismo para construir estruturas de dados apropriadas e MQTs dentro do DB2, podendo ser usadas para o processamento OLAP relacional. Visões de cubo oferecem suporte de modelagem para um esquema estrela relacional. Esse modelo é então usado para recomendar MQTs apropriadas, índices e definições MDC, a fim de melhorar o desempenho das consultas OLAP ao banco de dados. Além disso, visões de cubo podem tirar proveito do suporte nativo do DB2 para as operações **cube by** e **rollup**, de modo a gerar cubos agregados. Cube views é uma ferramenta que pode ser usada para a integração de perto do DB2 com fornecedores de OLAP, como Business Objects, Microstrategy e Cognos.

Além disso, o DB2 também oferece suporte OLAP multidimensional usando o servidor OLAP do DB2. O servidor OLAP do DB2 pode criar um data mart multidimensional a partir de um banco de dados DB2 básico, para ser analisado por técnicas OLAP. O mecanismo OLAP do produto Essbase é usado no servidor OLAP do DB2.

O DB2 Alphablox é um novo recurso que oferece capacidades de relatório e análise on-line, interativas. Um recurso bastante atraente do Alphablox é a capacidade de construir rapidamente novas formas de análise baseada na Web, usando uma técnica de bloco de montagem chamada *blox*.

Para os analíticos profundos, o DB2 Intelligent Miner oferece diversos componentes para modelagem, avaliação e visualização de dados. A exploração permite que os usuários realizem classificação, previsão, agrupamento, segmentação e associação em grandes conjuntos de dados.

Notas bibliográficas

A origem do DB2 pode ser acompanhada desde o projeto System R (Chamberlin e outros [1981]). Contribuições da IBM Research incluem áreas como processamento de transação (logging de escrita antecipada e algoritmos de recuperação ARIES) (Mohan *et al.* [1992]), processamento e

otimização de consulta (Starburst) (Haas et al. [1990]), processamento paralelo (DB2 Parallel Edition) (Baru et al. [1995]), suporte para banco de dados ativo (restrições, triggers) (Cochrane et al. [1996]), técnicas de consulta avançada e warehousing, como views materializadas (Zaharioudakis et al. [2000], Lehner et al. [2000]), agrupamento multidimensional (Pad-manabhan et al. [2003], Bhattacharjee et al. [2003]), recursos autonômicos (Zilio et al. [2004]) e suporte objeto-relacional (ADTs, UDFs) (Carey et al. [1999]). Os detalhes do processamento de consulta multiprocessador podem ser encontrados em Baru et al. [1995] ou nos guias DB2 Adminstration and Performance da documentação on-line do DB2.

A seguir está uma lista de materiais de referência sobre DB2. Os livros de Don Chamberlin oferecem uma boa revisão da SQL e recursos de programação (Chamberlin [1996], Chamberlin [1998]). Os livros mais antigos de C. J. Date *et al.* oferecem uma boa revisão dos recursos do DB2 Universal Database para OS/390 (Date [1989], Martin *et al.* [1989]). Os manuais do DB2 oferecem a visão definitiva de uma versão específica do DB2. A maior parte desses manuais está disponível on-line (documentação on-line do DB2). Livros recentes, como *DB2 for Dummies* (Zikopoulos *et al.* [2000]), *DB2 SQL Developer's Guide* (Sanders [2000]) e os *DB2 Administration Certification Guides* (Cook *et al.* [1999]) oferecem treinamento prático para uso e administração do DB2. Finalmente, a Prentice Hall está publicando uma série de livros sobre enriquecimento e certificação para vários aspectos do DB2.

Chamberlin [1998], Zikopoulos *et al.* [2004] e a biblioteca de documentação do DB2 oferecem uma descrição completa do suporte para SQL.

CAPÍTULO 29

Microsoft SQL Server

Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishann, Michael Rys, Florian Waads, Michael Zwilling
Microsoft

O Microsoft SQL Server é um sistema de gerenciamento de banco de dados relacional usado desde laptops e desktops até servidores de empresas, e também possui uma versão compatível, baseada no sistema operacional PocketPC, disponível para dispositivos manuais como o PocketPCs e scanners de código de barras. O SQL Server foi desenvolvido originalmente na década de 1980 na Sybase para sistemas UNIX e depois portado para os sistemas Windows NT pela Microsoft. Desde 1994, a Microsoft lança versões do SQL Server desenvolvidas independentemente da Sybase, que parou de usar o nome SQL Server no final da década de 1990. A última versão, o SQL Server 2005, está disponível nas edições Express, Standard e Enterprise e regionalizada para muitos idiomas ao redor do globo. Neste capítulo, o termo SQL Server se refere a todas essas edições do SQL Server 2005.

O SQL Server fornece serviços de duplicação entre várias cópias do SQL Server e com outros sistemas de banco de dados. Seu Analysis Services, uma parte integrante do sistema, inclui recursos de processamento analítico on-line (OLAP) e mineração de dados. O SQL Server fornece uma grande coleção de ferramentas gráficas e assistentes (wizards) que orientam administradores de banco de dados por meio de tarefas como configurar a realização de backups regulares, duplicar dados entre servidores e ajustar um banco de dados com vistas ao desempenho. Muitos ambientes de desenvolvimento de banco de dados aceitam o SQL Server, incluindo o Microsoft Visual Studio e produtos relacionados, em especial os produtos e serviços .NET.

Ferramentas de gerenciamento, projeto e consulta

O SQL Server fornece um conjunto de ferramentas para gerenciar todos os aspectos do desenvolvimento, consulta, configuração, teste e administração do SQL Server. A maioria dessas ferramentas se baseia no SQL Server Management Studio (anteriormente chamado de Enterprise Manager).

O Management Studio fornece um shell comum para administrar todos os serviços associados ao SQL Server, que inclui o Database Engine, o Analysis Services, o Reporting Services, o SQL Server Mobile e o Integration Services.

Desenvolvimento de banco de dados e ferramentas de banco de dados visuais

Enquanto está projetando um banco de dados, o administrador de banco de dados cria objetos de banco de dados como tabelas, colunas, chaves, índices, relacionamentos, restrições e views. Para ajudar a criar esses objetos, o SQL Server Management Studio fornece acesso a ferramentas de banco de dados visuais. Essas ferramentas oferecem três mecanismos para auxiliar no projeto de banco de dados: o Database Designer, o Table Designer e o View Designer.

O Database Designer é uma ferramenta visual que permite ao proprietário do banco de dados (ou a seus delegados) criar tabelas, colunas, chaves, índices, relacionamentos e restrições. Com essa ferramenta, o usuário pode interagir com objetos de banco de dados através de diagramas de banco de dados, que mostram graficamente a estrutura do banco de dados. O view Designer fornece uma ferramenta de consulta visual que permite ao usuário criar ou modificar views SQL valendo-se das capacidades de arrastar e soltar do Windows. A Figura 29.1 mostra uma view aberta pelo Management Studio.

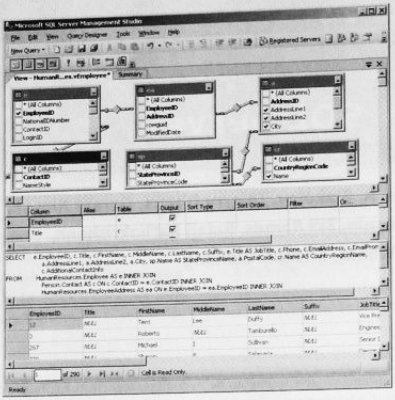


Figura 29.1 O View Designer, aberto para a view HumanResources.vEmployee.

Ferramentas de consulta e ajuste de banco de dados

O SQL Server Management Studio fornece várias ferramentas para auxiliar o processo de desempenho de aplicação. As consultas e os procedimentos armazenados podem ser desenvolvidos e testados usando o Query Editor integrado, que substitui o SQL Server Query Analyzer. O Query Editor aceita a criação e edição de scripts para T-SQL, SQLCMD, MDX, DMX, XMLA e SQL Server Mobile. Mais análises podem ser feitas usando o SQL Server Profiler. As recomendações de ajuste de banco de dados são fornecidas por uma terceira ferramenta, o Database Tuning Advisor.

Query Editor

O Query Editor integrado fornece uma interface gráfica com o usuário simples para executar consultas SQL e ver os resultados. O Query Editor também oferece uma representação gráfica do **showplan**, as etapas escolhidas pelo otimizador para a execução de consultas. O Query Editor é integrado ao Management Studio Object Explorer, que permite ao usuário arrastar e soltar nomes de objeto ou tabela para uma janela de consulta e ajuda a construir instruções **select**, **insert**, **update** e **delete** para qualquer tabela.

Um administrador ou desenvolvedor de banco de dados pode usar o Query Editor para:

- Analisar consultas: O Query Editor pode mostrar um plano de execução gráfico ou textual para qualquer consulta, bem como exibir estatísticas referentes ao tempo e recursos necessários para executar qualquer consulta.
- Formatar consultas SQL: Incluindo indexação e codificação de sintaxe por cores.
- Usar modelos para procedimentos armazenados, funções e instruções SQL básicas: O Management Studio vem com dezenas de modelos predefinidos para construir comandos DDL, que também podem ser definidos pelo próprio usuário.

A Figura 20.2 mostra o Management Studio com o Query Editor exibindo o plano de execução gráfica para uma consulta envolvendo uma junção de quatro tabelas e uma agregação.

SQL Profiler

O SQL Profiler é um utilitário gráfico que permite a administradores de banco de dados monitorar e registrar atividade de banco de dados do SQL Server Database Engine e Analysis Services. O SQL Profiler pode exibir todas as atividades do servidor em tempo real, ou pode criar filtros que focalizam as ações de determinados usuários, aplicações ou tipos de comandos. O SQL Profiler pode exibir qualquer instrução SQL ou procedimento armazenado enviado a

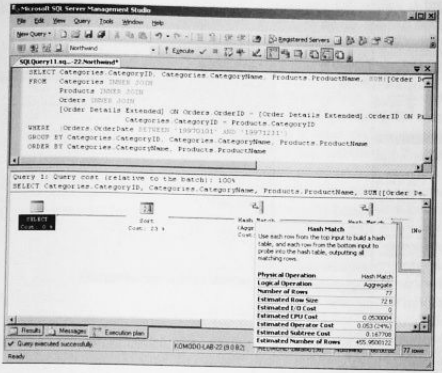


Figura 29.2 Um showplan para uma junção de quatro tabelas com agregação **group by.**

qualquer instância do SQL Server (se os privilégios de segurança permitirem), bem como dados de desempenho indicando o tempo que a consulta levou para ser executada, quanta CPU e E/S foram necessárias e o plano de execução que a consulta usou.

O SQL Profiler permite se aprofundar ainda mais no SQL Server para monitorar cada instrução executada como parte de um procedimento armazenado, cada operação de modificação de dados, cada bloqueio adquirido ou liberado, ou cada ocorrência de um arquivo de banco de dados crescendo automaticamente. Dezenas de eventos diferentes podem ser capturados e dezenas de itens de dados podem ser capturados para cada evento. O SQL Server efetivamente divide a funcionalidade de rastreamento em dois componentes distintos mas conectados. O SQL Profiler é a ferramenta de rastreamento do lado do cliente. Usando o SQL Profiler, um usuário pode escolher salvar os dados capturados em um arquivo ou tabela, além de exibi-los na Profiler User Interface (UI). A ferramenta Profiler exibe todos os eventos que atendem aos critérios de filtro no momento em que ocorrem. Uma vez que os dados são salvos, o SQL Profiler pode ler os dados salvos para fins de exibição ou análise.

No lado do servidor está a ferramenta de rastreamento SQL, que gerencia filas de eventos gerados por procedimentos de evento. Uma thread consumidora lê eventos das filas e os filtra antes de enviá-los para o processo que os requisitou. Os eventos são a unidade principal da atividade no que se refere ao rastreamento, e um evento pode ser qualquer coisa que ocorra dentro do SQL Server ou entre o SQL Server e um cliente. Por exemplo, criar ou descartar um objeto é um evento; executar um procedimento armazenado é um evento; assim como adquirir ou liberar um bloqueio e enviar um lote Transact-SQL de um cliente para o SQL Server. Há um conjunto de procedimentos de sistema armazenados para definir que eventos devem ser rastreados, que dados para que evento são de interesse e onde salvar as informações coletadas dos eventos. Filtros aplicados nos eventos podem reduzir a quantidade de informações coletadas e armazenadas.

O SQL Server garante que certas informações críticas sempre serão coletadas e poderão ser usadas como um mecanismo de auditoria útil. O SQL Server é certificado para segurança nível C2, e muitos dos eventos rastreáveis estão disponíveis apenas para aceitar requisitos de certificação C2.

O Database Tuning Advisor

As consultas e atualizações normalmente podem ser executadas muito mais rápido se um conjunto apropriado de índices estiver disponível. Projetar os melhores índices possíveis para as tabelas em um grande banco de dados é uma tarefa complexa; ela não só exige um completo conhecimento de como o SQL Server usa os índices e como o otimizador de consulta toma suas decisões, mas também de como os dados serão realmente usados pelas aplicações e consultas interativas. O SQL Server Database Tuning Advisor

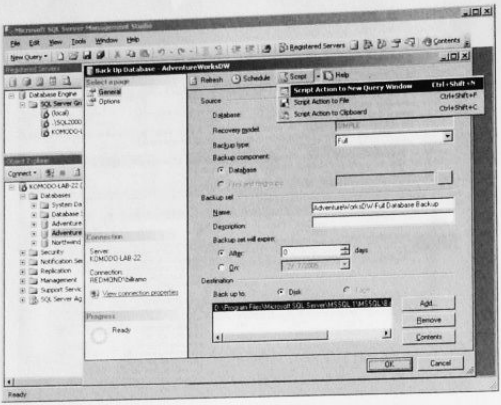


Figura 29.3 Interface do SQL Server Management Studio.

(DTA) (que substitui o SQL Server 2000 Index Tuning Wizard) é uma poderosa ferramenta para projetar os melhores índices possíveis e views indexadas (materializadas) com base nos workloads de consulta e atualização observados.

O DTA pode ajustar com a ajuda de vários bancos de dados e baseia suas recomendações em um workload que pode ser um arquivo dos eventos de rastreamento capturados, um arquivo das instruções SQL ou um arquivo de entrada XML. O SQL Profiler é projetado para capturar todas as instruções SQL submetidas por todos os usuários durante um determinado período de tempo. O DTA, então, olha os padrões de acesso a dados para todos os usuários, aplicações e tabelas, e faz recomendações equilibradas.

O SQL Server Management Studio

Além de fornecer acesso ao projeto de banco de dados e às ferramentas de banco de dados visuais e de ser fácil de usar, o SQL Server Management Studio aceita gerenciamento centralizado de todos os aspectos das instalações múltiplas do SQL Server Database Engine, do Analysis Services, do Reporting Services, do Integration Services e do SQL Server Mobile, incluindo segurança, eventos, alertas, agendamento, backup, configuração de servidor, ajuste, pesquisa de texto e duplicação. O SQL Server Management Studio permite que um administrador de banco de dados crie, modifique e copie esquemas de banco de dados do SQL Server e objetos como tabelas, views e acionadores. Como as instalações múltiplas do SQL Server podem ser organizadas em grupos e encadeadas como uma unidade, o SQL Server Management Studio pode gerenciar centenas de servidores simultaneamente.

Embora possa ser executado no mesmo computador do mecanismo SQL Server, o Management Studio oferece as mesmas capacidades de gerenciamento enquanto é executado em qualquer máquina Windows 2000 (ou posterior). Além disso, a eficiente arquitetura cliente-servidor do SQL Server torna conveniente usar as capacidades de acesso remoto (rede discada) do Windows para administração e gerenciamento.

O SQL Server Management Studio libera o administrador do banco de dados da necessidade de conhecer as etapas e a sintaxe específicas para completar uma tarefa. Ele oferece assistentes para guiar o administrador de banco de dados pelo processo de configurar e gerenciar uma instalação do SQL Server. A interface do Management Studio é mostrada na Figura 29.3 e ilustra como um script para backup de banco de dados pode ser criado diretamente de seus diálogos.

Variações e extensões da SQL

O SQL Server permite que desenvolvedores de aplicação escrevam lógica empresarial do lado do servidor usando a Transact-SQL ou uma linguagem de programação .NET como C#, Visual Basic, COBOL ou J++. A Transact-SQL é

uma linguagem de programação de banco de dados completa que inclui definição de dados e instruções de manipulação de dados, instruções interativas e condicionais, variáveis, procedimentos e funções. A Transact-SQL aceita a maioria das instruções e construções de consulta DDL e modificação de dados **obrigatórios** no padrão SQL-2003. Veja na próxima seção uma lista dos tipos de dados SQL-2003 aceitos. Além dos recursos obrigatórios, a Transact-SQL também aceita muitos recursos **opcionais** no padrão SQL-2003, como consultas recursivas, expressões de tabela comuns, funções definidas pelo usuário e operações relacionais (como **intersect** e **except**, entre outras).

Tipos de dados

O SQL Server aceita todos os tipos de dados escalares obrigatórios no padrão SQL-2003, exceto data e hora. Ele aceita o tipo de dados timestamp (também chamado de datetime), que permite armazenar os componentes de data e hora. O SQL Server também aceita a capacidade de atribuir identificação alternativa para tipos de sistema usando nomes fornecidos pelo usuário; esse recurso é semelhante, em funcionalidade, aos tipos distintos da SQL-2003, mas não totalmente concordante com eles.

Alguns tipos primitivos peculiares ao SQL Server incluem:

- Tipos de caractere longo e string binário de tamanho variável até $2^{31} - 1$ bytes (**text/ntext/image**, **varchar/nvarchar/varbinary(max)**). Os tipos de dados text/ntext/image exigem o uso de um textptr especializado que aja como um descritor ou um ponteiro para valores LOB. Os tipos de dados varchar/nvarchar/varbinary(max) possuem a mesma capacidade de byte de text/ntext/image, mas o modelo de programação é semelhante ao dos tipos de dados de caractere curto e string de byte.
- Um tipo XML, descrito na seção "Suporte a XML no SQL Server 2005", que é usado para armazenar dados XML dentro de uma coluna de tabela. O tipo XML pode, opcionalmente, ter uma *coleção de esquemas* XML associada especificando uma restrição de que as instâncias do tipo devem aderir a um dos tipos XML definidos na coleção de esquemas.
- Sql_variant é um tipo de dados escalar que pode conter valores de qualquer tipo escalar (exceto tipos de caractere longo e binário e sql-variant). Esse tipo é usado por aplicações que precisam armazenar dados cujo tipo não pode ser previsto no tempo de definição de dados. Sql_variant também é o tipo de uma coluna formada da execução de um operador relacional **unpivot** (veja a próxima seção). Internamente, o sistema controla o tipo original dos dados. É possível filtrar, juntar e classificar sob colunas sql_variant. A função de sistema **sql_variant_property** retorna detalhes sobre os dados reais armazenados em uma coluna do tipo sql_variant, incluindo informações do tipo base e tamanho.

Além disso, o SQL Server aceita um tipo de tabela e um tipo de cursor que não pode ser usado como colunas em uma tabela, mas pode ser usado na linguagem Transact-SQL como variáveis:

- Um tipo **table** permite que uma variável armazene um conjunto de linhas. Uma instância desse tipo é usada principalmente para armazenar resultados temporários em um procedimento armazenado ou como o valor de retorno de uma função com valor de tabela. Uma variável table age como uma variável local. Ela tem um escopo bem definido, que é a função, o procedimento armazenado ou o lote em que é declarada. Dentro de seu escopo, uma variável table pode ser usada como uma tabela comum. Ela pode ser aplicada em qualquer lugar em que uma tabela ou expressão de tabela possa ser usada nas instruções **select**, **insert**, **update** e **delete**.
- Um tipo **cursor** permite referências a um objeto cursor. O tipo cursor pode ser usado para declarar variáveis ou argumentos de entrada/saída de modo a referenciar cursores por chamadas de rotina.

Melhorias na linguagem de consulta

Além dos operadores relacionais SQL, como **inner join** e **outer join**, o SQL Server aceita os operadores relacionais **pivot**, **unpivot** e **apply**.

- **pivot** é um operador que converte a forma de seu conjunto resultado de entrada de duas colunas que representam pares nome-valor para múltiplas colunas, uma para cada nome da entrada. A coluna de nome da entrada é a coluna pivô. O usuário precisa indicar que nomes transportar da entrada para colunas individuais na saída. Considere a tabela *VendasMensais*(*ID_Produto*, *Mês*, *Quant_Vendas*). A seguinte consulta usando o operador **pivot** retorna a *Quant_Vendas* para cada um dos meses Jan, Fev e Mar como colunas separadas. Observe que o operador **pivot** também realiza uma agregação implícita em todas as outras colunas na tabela e uma agregação explícita na coluna pivô.

```
select*
    from VendasMensais pivot(sum(Quant_Vendas)
    for mês in('Jan', 'Fev', 'Mar')) T
```

A operação inversa de **pivot** é **unpivot**.

- O operador **apply** é um operador binário que toma duas entradas com valor de tabela, das quais a entrada direita normalmente é uma chamada de função com valor de tabela que toma como argumentos uma ou mais colunas da entrada esquerda. As colunas geradas pelo operador são a união das colunas de suas duas entradas. O operador **apply** pode ser usado para avaliar sua entrada direita para cada coluna de sua entrada esquerda e realizar uma operação **union all** das linhas entre todas essas avaliações. Existem dois tipos de operador **apply** semelhantes a **join**: **cross** e **outer**. Os dois tipos diferem na maneira como manipulam o caso da entrada direita produzindo um conjunto resultado vazio. No caso de **cross apply**, isso faz com que a linha correspondente da entrada esquerda não apareça no resultado. No caso de **outer apply**, a linha aparece a partir da entrada esquerda com valores NULL para as colunas na entrada direita. Considere uma função com valor de tabela chamada *EncontraSubords*, que toma como entrada o ID de um dado funcionário e retorna o conjunto dos funcionários subordinados direta ou indiretamente a esse funcionário em uma organização. A seguinte consulta chama essa função para o gerente de cada departamento da tabela **Departamentos**:

```
select *
from Departamentos D cross apply
EncontraSubords(D.IDGerente)
```

Rotinas

Os usuários podem escrever rotinas que são executadas dentro do processo de servidor como funções ou tabelas escalares, procedimentos armazenados e acionadores, usando a Transact-SQL ou uma linguagem .NET. Todas essas rotinas são definidas para o banco de dados usando a instrução DDL **create**[**function**, **procedure**, **trigger**] correspondente. As funções escalares podem ser usadas em qualquer expressão escalar dentro de uma instrução SQL DML ou DDL. As funções com valor de tabela podem ser usadas em qualquer lugar em que uma tabela é permitida em uma instrução **select**. As funções com valor de tabela da Transact-SQL cujo corpo contenha uma única instrução SQL **select** são tratadas como uma view (expandida em linha) na consulta que referencia a função. Como as funções com valor de tabela permitem argumentos de entrada, as funções com valor de tabela em série podem ser consideradas views parametrizadas.

Views indexadas

Além das views tradicionais conforme definidas na ANSI SQL, o SQL Server aceita views indexadas (materializadas). As views indexadas podem melhorar substancialmente o desempenho das consultas complexas de suporte a decisão que recuperam grandes números de linhas de tabela base e agregam grandes quantidades de informação em somas, contagens e médias concisas. O SQL Server aceita a criação de um índice agrupado em uma view e, subsequentemente, qualquer número de índices não agrupados. Uma vez que uma view é indexada, o otimizador pode usar seus índices em consultas que referenciam a view ou suas tabelas base. As consultas existentes podem se beneficiar da melhor eficiência da recuperação de dados diretamente da view indexada sem precisarem ser reescritas para referenciar a view. As instruções update para as tabelas base da view são propagadas automaticamente para as views indexadas.

Views atualizáveis e acionadores

Em geral, as views podem ser o destino das instruções **update**, **delete** e **insert** se a modificação dos dados se aplica apenas a uma das tabelas base da view. As atualizações a views particionadas podem ser propagadas para várias tabelas base. Por exemplo, a seguinte instrução **update** aumentará os preços da editora "0736" em 10%.

```
update viewtitulo
set preço = preço * 1.10
where id_edit = '0736'
```

Para modificações de dados que afetem mais de uma tabela base, a view pode ser atualizada se houver um acionador **instead** definido para a operação: acionadores **instead** para operações **insert**, **update** ou **delete** podem ser definidos em uma view, a fim de especificar as atualizações que precisam ser realizadas nas tabelas base de modo a refletir as modificações correspondentes na view.

Os acionadores são procedimentos Transact-SQL ou .NET que são executados automaticamente quando uma instrução DML (**update**, **insert** ou **delete**) ou DDL é emitida em uma tabela base ou view. Os acionadores são mecanismos que permitem a imposição de lógica empresarial automaticamente quando dados são modificados ou quando instruções DDL são executadas. Os acionadores podem estender a lógica de verificação de integridade das restrições declarativas, padrões e regras, embora as restrições declarativas devam ser usadas preferivelmente sempre que forem suficientes.

Os acionadores podem ser classificados em acionadores DML ou DDL, dependendo do tipo de evento que dispara o acionador. Os acionadores DML são definidos no nível de uma tabela ou view que está sendo modificada. Os acionadores DDL são definidos no nível de um banco de dados inteiro para uma ou mais instruções DDL, como **create table** e **drop procedure**.

Os acionadores podem ser classificados em acionadores **after** e **instead**, conforme o momento em que são chamados em relação à ação que dispara o acionador. Os acionadores **after** são executados após a instrução acionadora e as restrições declarativas subseqüentes serem impostas. Os acionadores **instead** são executados no lugar da ação de acionamento. Os acionadores **instead** podem ser imaginados como análogos aos acionadores **before**, mas realmente substituem a ação acionadora. No SQL Server, os acionadores DML **after** podem ser definidos apenas em tabelas base, enquanto os acionadores DML **instead** podem ser definidos em tabelas base ou em views. Os acionadores **instead** permitem que praticamente qualquer view se torne atualizável. Os acionadores DDL **instead** podem ser definidos em qualquer instrução DDL.

Armazenamento e indexação

No SQL Server, um banco de dados se refere a uma coleção de arquivos que contêm dados e são aceitos por um único log de transação. O banco de dados é a unidade de administração principal no SQL Server e também fornece um container para estruturas físicas, como tabelas e índices, e para estruturas lógicas, como restrições e views.

Filegroups

Para gerenciar o espaço de maneira eficiente em um banco de dados, o conjunto de arquivos de dados em um banco de dados é dividido em grupos chamados filegroups. Cada filegroup contém um ou mais arquivos de sistema operacional.

Todo banco de dados possui pelo menos um filegroup conhecido como o filegroup primário. Esse filegroup contém todos os metadados para o banco de dados nas tabelas de sistema. O filegroup primário também pode armazenar dados do usuário.

Se outros filegroups definidos pelo usuário tiverem sido criados, um usuário pode controlar explicitamente o posicionamento de tabelas individuais, de índices ou de colunas de objeto grande de uma tabela colocando-os em um filegroup. Por exemplo, o usuário pode escolher armazenar uma tabela no filegroupA, seu índice não agrupado no filegroupB e as colunas de objeto grande da tabela no filegroupC. Colocar essas tabelas e índices em diferentes filegroups permite que o usuário controle o uso dos recursos de hardware (ou seja, discos e o subsistema de E/S). Um filegroup é sempre considerado o filegroup padrão; inicialmente, o filegroup padrão é o filegroup primário, mas qualquer filegroup definido pelo usuário pode receber a *propriedade padrão*. Se uma tabela ou índice não for colocado especificamente em um filegroup, ela(ele) será criada(o) no filegroup padrão.

Gerenciamento de espaço dentro de filegroups

Um dos principais objetivos dos filegroups é permitir um gerenciamento eficiente do espaço. Todos os arquivos de dados são divididos em unidades com tamanho fixo de 8 kilobytes, chamadas **páginas**. O sistema de alocação é responsável por alocar essas páginas em tabelas e índices. O objetivo do sistema de alocação é minimizar a quantidade de espaço desperdiçado e, ao mesmo tempo, manter a quantidade de fragmentação no banco de dados em um mínimo a fim de garantir um bom desempenho de varredura. Para atingir esse objetivo, o gerenciador de alocação normalmente aloca e desaloca todas as páginas em unidades de oito páginas contíguas chamadas **extensões**.

O sistema de alocação gerencia essas extensões através de vários bitmaps. Esses bitmaps permitem que o sistema de alocação encontre rapidamente uma página ou uma extensão para alocação. Esses bitmaps também são usados quando uma varredura de tabela ou índice completa é executada. A vantagem de usar bitmaps baseados em alocação para varredura é que isso permite travessias de ordem de disco de todas as extensões pertencentes ao nível de folha de uma tabela ou índice, o que melhora significativamente o desempenho de varredura.

Se houver mais de um arquivo em um filegroup, o sistema de alocação aloca extensões para qualquer objeto nesse filegroup usando um algoritmo de "preenchimento proporcional". Cada arquivo é preenchido na proporção da quantidade de espaço livre nesse arquivo comparado a outros arquivos. Isso preenche todos os arquivos em um filegroup aproximadamente na mesma proporção e permite que o sistema utilize todos os arquivos no filegroup de maneira uniforme.

Uma das decisões mais importantes ao configurar um banco de dados é determinar o tamanho que ele deve ter. O SQL Server permite que arquivos de dados mudem de tamanho depois que o banco de dados é criado. O usuário pode até mesmo escolher fazer o arquivo de dados crescer automaticamente se o banco de dados estiver esgotando o espaço. Portanto, o usuário pode configurar o banco de dados para uma aproximação razoável do tamanho esperado, mas definir os arquivos de banco de dados para crescerem e se ajustarem ao padrão de uso, se a aproximação inicial estiver errada. O SQL Server também permite que os arquivos sejam reduzidos. Para reduzir um arquivo de dados, o SQL Server move todos os dados do final físico do arquivo para um ponto próximo do início do arquivo e, depois, efetivamente reduz o arquivo, liberando espaço novamente para o sistema operacional.

Tabelas

O SQL Server aceita organizações em heap e agrupadas para tabelas. Em uma tabela organizada em heap, a aloca-

ção de cada linha da tabela é determinada inteiramente pelo sistema e, de modo algum, é especificada pelo usuário. As linhas de um heap possuem um identificador fixo conhecido como ID de linha (RID), e esse valor nunca muda, a menos que o arquivo seja reduzido e a linha seja movida. Se a linha se tornar grande a ponto de não caber na página em que estava originalmente inserida, o registro é movido para um local diferente, mas um stub direto é deixado no lugar original de modo que o registro ainda possa ser encontrado usando seu RID original.

Em uma organização de índice agrupada para uma tabela, as linhas da tabela são armazenadas em uma árvore B^+ classificada pela chave de agrupamento do índice. A chave de índice agrupado também age como o identificador único para cada linha. A chave para um índice agrupado pode ser definida para ser não única, caso em que o SQL Server acrescenta uma coluna oculta adicional para tornar a chave única. O índice agrupado também serve como uma estrutura de pesquisa para identificar uma linha da tabela com uma determinada chave ou para varrer um conjunto de linhas da tabela com chaves dentro de uma certa faixa. Um índice agrupado é o tipo mais comum de organização de tabela.

Índices

O SQL Server também aceita índices de árvore B^+ secundários (não agrupados). As consultas que se referem apenas às colunas que estão disponíveis por meio de índices secundários são processadas recuperando páginas do nível de folha dos índices sem precisar recuperar dados do índice ou heap agrupado. Os índices não agrupados em uma tabela com um índice agrupado contêm as colunas-chave do índice agrupado. Portanto, as linhas de índice agrupado podem ser movidas para uma página diferente (via divisões, desfragmentação ou mesmo reconstruções de índice) sem exigir mudanças nos índices não agrupados.

O SQL Server aceita a adição de colunas calculadas em uma tabela. Uma coluna calculada é aquela cujo valor é uma expressão, normalmente baseada no valor de outras colunas nessa linha. O SQL Server permite que o usuário construa índices secundários nas colunas calculadas.

Partições

O SQL Server aceita particionamento de faixa em tabelas e índices não agrupados. Um índice particionado é composto de várias árvores B^+, uma por partição. Uma tabela particionada sem um índice (um heap) é composta de múltiplos heaps, um por partição. Para maior brevidade, nos referimos apenas aos índices particionados (agrupados ou não) e ignoramos os heaps no restante deste estudo.

Particionar um índice grande oferece a um administrador mais flexibilidade para gerenciar o armazenamento para o índice e pode melhorar o desempenho de consulta porque as partições agem como um índice de granularidade fina.

O particionamento para um índice é especificado fornecendo uma função de particionamento e um esquema de particionamento. Uma função de particionamento mapeia o domínio de uma coluna de particionamento (qualquer coluna no índice) para partições numeradas de 1 a N. Um esquema de particionamento mapeia os números de partição produzidos por uma função de particionamento para filegroups específicos em que as partições estão armazenadas.

Construção de índice on-line

Construir novos índices e reconstruir índices existentes em uma tabela pode ser realizado on-line, ou seja, enquanto as operações select/insert/delete/update estão sendo executadas na tabela. A criação de um novo índice ocorre em três fases. A primeira fase é simplesmente criar uma árvore B^+ vazia para o novo índice com o catálogo mostrando que o novo índice está disponível para operações de manutenção. Ou seja, o novo índice precisa ser mantido por todas as operações insert/delete/update subseqüentes, mas ele não está disponível para consultas. A segunda fase consiste na varredura da tabela a fim de recuperar as colunas de índice para cada linha, classificando as linhas e as inserindo na nova árvore B^+. Essas inserções precisam ser cuidadosas para interagir com as outras linhas na nova árvore B^+ colocadas lá pelas operações de manutenção de índice das atualizações na tabela base. A varredura é uma varredura de instantâneo que, sem bloqueio, garante que a varredura veja a tabela inteira com apenas os resultados das transações confirmadas no início da varredura. Isso é conseguido usando a tecnologia de isolamento de instantâneo descrita na seção "Transações". A última fase da construção de índice envolve atualizar o catálogo para indicar que a construção do índice está completa e o índice está disponível para consultas.

Varreduras e leitura antecipada

A execução de consultas no SQL Server pode envolver vários modos de varredura diferentes nas tabelas e índices originais. Esses modos incluem varreduras ordenadas e não ordenadas, varreduras seriais e paralelas, varreduras unidirecionais e bidirecionais, varreduras para a frente e para trás, varreduras de tabela ou índice inteiro e varreduras de faixa ou filtradas.

Cada um dos modos de varredura possui um mecanismo de leitura antecipada, que tenta manter a varredura à frente das necessidades da execução da consulta, a fim de reduzir os overheads de busca e latência e utilizar o tempo ocioso

do disco. O algoritmo de leitura antecipada do SQL Server usa o conhecimento do plano de execução de consulta para controlar a leitura antecipada e garantir que apenas os dados realmente necessários pela consulta sejam lidos. Além disso, a quantidade de leitura antecipada é automaticamente dimensionada de acordo com o tamanho do pool de buffers, a quantidade de E/S que o subsistema de disco pode sustentar e a velocidade em que os dados estão sendo consumidos pela execução da consulta.

Processamento e otimização de consulta

O processador de consulta do SQL Server é baseado em uma estrutura extensível que permite rápida incorporação de novas técnicas de execução e otimização. Qualquer consulta SQL pode ser expressa como uma árvore de operadores da álgebra relacional estendida do SQL Server. Abstraindo operadores dessa álgebra em *repetidores*, a execução da consulta encapsula algoritmos de processamento de dados como unidades lógicas que se comunicam umas com as outras usando a interface GetNextRow( ). Começando com uma árvore de consulta inicial, o otimizador de consulta do SQL Server gera alternativas usando transformações de árvore e estima seu custo de execução levando em conta o comportamento do repetidor e os modelos estáticos para derivar seletividades.

Visão geral do processo de otimização

As consultas complexas apresentam oportunidades de otimização significativas que exigem a reordenação de operadores através dos limites do bloco e a seleção de planos unicamente na base dos custos estimados. Para procurar essas oportunidades, o otimizador de consulta do SQL Server se desvia dos métodos de otimização de consulta tradicionais usados em outros sistemas comerciais em favor de uma estrutura mais geral e puramente algébrica, baseada no protótipo do otimizador Cascades. A otimização de consulta é parte do processo de compilação de consulta, que consiste em quatro etapas:

- **Analise/vinculação**. O analisador resolve nomes de tabela e coluna usando os catálogos. O SQL Server utiliza um cache de plano para evitar a otimização repetida de consultas idênticas ou estruturalmente semelhantes. Se nenhum plano estiver disponível em cache, uma árvore de operadores inicial é gerada. A árvore de operador é simplesmente uma combinação de operadores relacionais e não está restrita por conceitos como blocos de consulta ou tabelas derivadas, o que normalmente obstrui a otimização.
- **Simplificação/normalização.** O otimizador aplica regras de simplificação na árvore de operador para obter uma forma simplificada e normal. Durante a simplificação, o otimizador determina e carrega estatísticas necessárias para a estimação de cardinalidade.
- **Otimização baseada em custo**. O otimizador aplica regras de exploração e implementação para gerar alternativas, estima o custo da execução e escolhe o plano com a previsão de custo mais baixa. As regras de exploração implementam a reordenação para um conjunto extensivo de operadores, incluindo reordenação de junção e agregação. As regras de implementação apresentam alternativas de execução, como junção de mesclagem e junção de hash.
- **Preparação de plano**. O otimizador cria estruturas de execução de consulta para o plano selecionado.

Para obter melhores resultados, a otimização baseada em custo do SQL Server não é dividida em fases que otimizam diferentes aspectos da consulta independentemente; além disso, ela não se limita a uma única dimensão, como a enumeração de junção. Em vez disso, uma coleção de regras de transformação define o espaço de interesse, e a estimativa de custo é usada uniformemente para selecionar um plano eficiente.

Simplificação de consulta

Durante a simplificação, apenas as transformações com a garantia de gerar substitutos menos onerosos são aplicadas. O otimizador envia selects pela árvore de operadores o mais longe possível; ele verifica contradições em predicados, levando em conta restrições declaradas. Ele usa as contradições para identificar subexpressões que possam ser removidas da árvore. Um cenário comum é a eliminação de desvios **union** que recuperam dados de tabelas com diferentes restrições.

Diversas regras de simplificação são *dependentes do contexto*; isto é, a substituição é válida apenas no contexto da utilização da subexpressão. Por exemplo, uma junção externa pode ser simplificada para uma junção interna se uma operação de filtro posterior for descartar regras de inadequação que foram preenchidas com **null**. Outro exemplo é a eliminação de junções em chaves estrangeiras, que não precisa ser realizada se não houver usos posteriores das colunas da tabela referenciada. Um terceiro exemplo é o contexto da insensibilidade de duplicata, que especifica que distribuir uma ou mais cópias de uma linha não afeta o resultado da consulta. As subexpressões sob semijunções e sob **distinct** são insensíveis para duplicata, o que permite transformar **union** em **union all**.

Para agrupamento e agregação, é usado o operador *GbAgg*, que cria grupos e aplica opcionalmente uma função agregada em cada grupo. A remoção de duplicata, expressa

na SQL pela palavra-chave **distinct**, é simplesmente um *GbAgg* sem funções agregadas para calcular. Durante a simplificação, as informações sobre chaves e dependências funcionais são usadas para reduzir colunas de agrupamento.

As subconsultas são normalizadas removendo especificações de consulta correlata e usando alguma variante de junção no lugar. Remover correlações não é uma "estratégia de execução de subconsulta", mas simplesmente uma etapa de normalização. Portanto, uma variedade de estratégias de execução é considerada durante a otimização baseada em custo.

Reordenação e otimização baseada em custo

No SQL Server, as transformações são totalmente integradas com a geração baseada em custo e a seleção de planos de execução. O otimizador de consulta do SQL Server inclui aproximadamente 350 regras de transformação lógicas e físicas. Além da reordenação de junção interna, o otimizador de consulta emprega transformações de reordenação para os operadores de junção externa, semijunção e antisemijunção da álgebra relacional padrão (com duplicatas, para a SQL). O operador *GbAgg* é reordenado também, movendo-o para baixo das junções onde possível. A agregação parcial, ou seja, introduzir um novo *GbAgg* com agrupamento em um superconjunto das colunas de um *GbAgg* subseqüente, é considerada abaixo das junções e do **union all** e também nos planos paralelos. Veja as referências fornecidas nas notas bibliográficas para obter detalhes.

A execução correlata é considerada durante a exploração do plano, sendo que o caso mais simples é a junção de consulta de índice. O SQL Server modela execução correlata como um único operador algébrico, chamado **apply**, que opera em uma tabela T e uma expressão relacional parametrizada $E(t)$. O operador **apply** executa E para cada linha de T, o que fornece valores de parâmetro. A execução correlata é considerada como uma execução alternativa, independente do uso das subconsultas na formulação SQL original. Ela é uma estratégia bastante eficaz quando a tabela T é pequena e os índices aceitam execução parametrizada eficiente de $E(t)$. Além disso, consideramos a redução no número de execuções de $E(t)$ quando existem valores de parâmetro duplicados, por meio de duas técnicas: classificar T sob valores de parâmetro de modo que um único resultado de $E(t)$ seja reutilizado enquanto o valor de parâmetro permanece igual, ou então usar uma tabela de hash que controla o resultado de $E(t)$ para (algum subconjunto de) valores de parâmetro anteriores.

Algumas aplicações selecionam linhas na base de algum resultado agregado para seu grupo. Por exemplo, "Encontre os clientes cujo saldo é mais do que o dobro da média para seu segmento de mercado." A formulação SQL exige uma autojunção. Durante a exploração, esse padrão é detectado e a execução por segmento é considerada como uma alternativa à autojunção.

A utilização de view materializada também é considerada durante a otimização baseada em custo. A correspondência de view interage com a reordenação de operador em que a utilização pode não ser aparente até que alguma outra reordenação tenha ocorrido. Quando uma view é encontrada correspondendo a alguma subexpressão, a tabela que contém o resultado de view é acrescentada como uma alternativa para a expressão correspondente. Dependendo da distribuição de dados e dos índices disponíveis, ela pode ou não ser melhor do que a expressão original – a seleção será baseada na estimativa de custo.

Para estimar o custo de execução de um plano, o modelo leva em conta o número de linhas esperadas para o processo, o que chamamos de meta de linhas, bem como o número de vezes que uma subexpressão é executada. A meta de linhas pode ser menor do que a cardinalidade estimada, em casos como *Apply/semijoin*. *Apply/semijoin* gera a saída da linha t de T logo que uma única linha seja produzida por $E(t)$ (isto é, ele testa **exists** $E(t)$). Portanto, a meta de linhas da saída de $E(t)$ é 1, e as metas de linhas das subárvores de $E(t)$ são calculadas para essa meta de linhas para $E(t)$ e usadas em estimativa de custo.

Planos de atualização

Os planos de atualização otimizam a manutenção de índices, verificam restrições, aplicam ações em cascata e mantêm views materializadas. Para a manutenção de índices, em vez de tomar cada linha e manter todos os índices para ela, os planos de atualização aplicam modificações por índice, classificando linhas e aplicando a operação update na ordem-chave. Isso minimiza a E/S aleatória, especialmente quando o número de linhas a serem atualizadas é grande. As restrições são tratadas por um operador **assert**, que executa um predicado e gera um erro se o resultado for **false**. As restrições referenciais são definidas por predicados **exists**, que, por sua vez, se tornam semijunções e são otimizadas considerando todos os algoritmos de execução.

O problema do Halloween é resolvido usando opções baseadas em custo. O problema do Halloween se refere à seguinte anomalia: suponha que um índice de salário é lido na ordem crescente e os salários estão sendo aumentados em 10%. Como um resultado da atualização, as linhas serão movidas para a frente no índice e serão encontradas e atualizadas novamente, levando a um loop infinito. Uma maneira de resolver esse problema é separar o processamento em duas fases: A primeira lê todas as linhas que serão atualizadas e faz uma cópia delas em algum local temporário, de-

pois, lê desse local e aplica todas as atualizações. Outra alternativa é ler de um índice diferente em que as linhas não serão movidas como resultado da atualização. Alguns planos de execução fornecem separação de fase automaticamente, se eles classificarem ou construírem uma tabela de hash nas linhas a serem atualizadas. No otimizador do SQL Server, a proteção de Halloween é modelada como uma propriedade dos planos. São gerados vários planos que fornecem a propriedade necessária, e um deles é selecionado com base no custo de execução estimado.

Análise de dados em tempo de otimização

A SQL foi a pioneira na utilização de técnicas para realizar coleta de estatística como parte de uma otimização constante. O cálculo das estimativas de tamanho do resultado é baseado nas estatísticas para colunas usadas em uma determinada expressão. Essas estatísticas consistem em histogramas max-diff nos valores de coluna e vários contadores que capturam densidades e tamanhos de linha, entre outros. Os administradores de banco de dados podem criar estatísticas explicitamente usando a sintaxe SQL estendida.

Se, no entanto, nenhuma estatística estiver disponível para uma determinada coluna, o otimizador do SQL Server suspende a otimização constante e coleta estatísticas conforme necessário. Assim que as estatísticas são calculadas, a otimização original é retomada, impondo as estatísticas recém-criadas. A otimização de consultas subsequentes reutiliza estatísticas geradas anteriormente. Em geral, após um curto período de tempo, estatísticas para colunas frequentemente usadas são criadas e interrupções para coletar novas estatísticas se tornam raras. Controlando o número de linhas modificadas em uma tabela, uma medida de caducidade é mantida para todas as estatísticas afetadas. Quando a caducidade excede um determinado limite, as estatísticas são recalculadas e os planos em cache são recompilados levando-se em conta as distribuições de dados modificadas.

O SQL Server 2005 pode realizar cálculo estatístico automático de forma assíncrona. Isso evita tempos de compilação potencialmente longos causados pela coleta síncrona de estatística. A otimização que aciona o cálculo da estatística usa estatística potencialmente desatualizada. Entretanto, consultas subseqüentes são capazes de impor estatística recalculada. Isso permite alcançar um equilíbrio aceitável entre o tempo gasto na otimização e a qualidade do plano de consulta resultante.

Pesquisa parcial e heurística

Os otimizadores de consulta baseados em custo enfrentam o problema da explosão do espaço de pesquisa porque as aplicações emitem consultas envolvendo dezenas de tabelas. Para resolver isso, o SQL Server usa vários estágios de otimização, cada um com transformações de consulta para explorar regiões cada vez maiores do espaço de pesquisa.

Existem transformações simples e completas, voltadas para a otimização exaustiva, bem como transformações inteligentes que implementam várias heurísticas. As transformações inteligentes geram planos que se encontram bastante separados no espaço de pesquisa, enquanto as transformações simples exploram os vizinhos. Os estágios de otimização aplicam um misto de tipos de transformações, enfatizando primeiro as transformações inteligentes e depois transicionando para as transformações simples. Resultados ótimos nas subárvores são preservados, de modo que os estágios posteriores possam tirar proveito dos resultados gerados anteriormente. Cada estágio precisa equilibrar técnicas de geração de plano opostas:

- **Geração exaustiva de alternativas:** para gerar o espaço completo, o otimizador usa transformações completas, locais e não redundantes – uma regra de transformação equivalente a uma seqüência de transformações mais primitivas apenas introduz overhead adicional.
- **Geração heurística de candidatos:** um conjunto de candidatos interessantes (selecionados com base no custo estimado) provavelmente estará bastante distante em termos de regras de transformação primitiva. Aqui, as transformações desejáveis são incompletas, globais e redundantes.

A otimização pode ser terminada em qualquer ponto após o primeiro plano ter sido gerado. Essa terminação é baseada no custo estimado do melhor plano encontrado e no tempo já gasto na otimização. Por exemplo, se uma consulta exige apenas pesquisar algumas linhas em alguns índices, um plano muito barato será produzido rapidamente nos primeiros estágios, terminando a otimização. Esse método permitiu incluir nova heurística facilmente ao longo do tempo, sem comprometer a seleção dos planos baseada em custo ou a exploração exaustiva do espaço de pesquisa, quando apropriado.

Execução de consulta

Os algoritmos de execução aceitam processamento baseado em classificação e baseado em hash, e suas estruturas de dados são projetadas para otimizar o uso do cache do processador. As operações de hash suportam agregação e junção básicas, com diversas otimizações, extensões e ajuste de dinâmica para distorção de dados. A operação flow-distinct é uma variante de hash-distinct, em que a saída das linhas é gerada inicialmente, assim que um novo valor distinto é encontrado, em vez de esperar o processo

completar a entrada. Esse operador é eficiente para consultas que usam **distinct** e exigem apenas algumas linhas, digamos, usando a construção **top n**. Planos correlatos especificam a execução de $E(t)$, geralmente incluindo alguma pesquisa de índice baseada no parâmetro, para cada linha t de uma tabela T. A *pré-busca assíncrona* permite emitir múltiplas requisições de pesquisa de índice para o mecanismo de armazenamento. Ela é implementada desta forma: uma requisição de consulta de índice sem bloqueio é feita para uma linha t de T; depois, t é colocada em uma fila de pré-busca. As linhas são tiradas da fila e usadas por **apply** para executar $E(t)$. A execução de $E(t)$ não exige que os dados já estejam no pool de buffers, mas ter operações de pré-busca satisfatórias maximiza a utilização do hardware e melhora o desempenho. O tamanho da fila é determinado dinamicamente como uma função dos acessos ao cache. Se nenhuma ordenação é necessária nas linhas de saída de **apply**, as linhas da fila podem ser tiradas fora de ordem, para minimizar a espera na E/S.

A execução paralela é implementada pelo operador **exchange**, que gerencia threads múltiplas, partições ou dados de difusões, e alimenta os dados para vários processos. O otimizador de consulta decide trocar o posicionamento com o operador **exchange** com base no custo estimado. O grau de paralelismo é determinado dinamicamente em tempo de execução, de acordo com a utilização atual do sistema.

Os planos de índice são formados das partes descritas anteriormente. Por exemplo, consideramos o uso de uma junção de índice para resolver conjunções de predicados (ou união de índice, para disjunções), de uma maneira baseada em custo. Essa junção pode ser feita em paralelo, usando qualquer um dos algoritmos de junção do SQL Server. Também consideramos juntar índices com o único fim de montar uma linha com o conjunto de colunas necessárias em uma consulta, o que, às vezes, é mais rápido do que varrer a tabela base. Tomar IDs de registro de um índice secundário e localizar a linha correspondente em uma tabela base é, na verdade, equivalente a realizar junção de pesquisa de índice. Para isso, usamos nossas técnicas de execução correlatas genéricas, como pré-busca assíncrona.

A comunicação com o mecanismo de armazenamento é feita pelo OLE-DB, que permite acessar outros provedores de dados que implementam essa interface. O OLE-DB é o mecanismo usado para consultas distribuídas e remotas, que são controladas diretamente pelo processador de consulta. Os provedores de dados são categorizados de acordo com a faixa de funcionalidade que eles oferecem, variando de simples provedores de conjunto de linhas sem capacidades de indexação a provedores com total suporte a SQL.

Concorrência e recuperação

Os subsistemas de transação, logging, bloqueio e recuperação do SQL Server realizam as propriedades ACID esperadas de um sistema de banco de dados.

Transações

No SQL Server, todas as instruções são atômicas e as aplicações podem especificar vários níveis de isolamento para cada instrução. Uma única transação pode incluir instruções que não apenas selecionam, inserem, excluem ou atualizam registros, mas também criam ou descartam tabelas, constroem índices e importam dados em massa. As transações podem abranger bancos de dados em servidores remotos. Quando as transações são espalhadas entre servidores, o SQL Server usa um serviço do sistema operacional Windows chamado Microsoft Distributed Transaction Coordinator (MS DTC) para realizar processamento de confirmação em duas fases. O MS DTC aceita o protocolo de transação XA e, juntamente com o OLE-DB, fornece a fundação para as transações ACID entre sistemas heterogêneos.

O controle de concorrência baseado em bloqueio é o padrão para o SQL Server. O SQL Server também oferece controle de concorrência seguro para cursores. O controle de concorrência seguro se baseia na suposição de que os conflitos de recurso entre vários usuários são improváveis (mas não impossíveis), e permite que as transações sejam executadas sem bloquear quaisquer recursos. Apenas quando tenta mudar os dados o SQL Server verifica os recursos para determinar se quaisquer conflitos ocorreram. Se um conflito ocorrer, a aplicação precisa ler os dados e tentar a mudança novamente. As aplicações podem escolher detectar mudanças comparando valores ou verificando uma coluna de versão de linha especial em uma linha.

O SQL Server aceita os níveis de isolamento SQL de leitura não confirmada, leitura confirmada, leitura repetível e leitura serializável. Leitura confirmada é o nível padrão. Além disso, o SQL Server aceita dois níveis de isolamento baseados em instantâneo.

- **Instantâneo**: especifica que os dados lidos por qualquer instrução em uma transação sejam a versão transacionalmente consistente dos dados que existia no início da transação. A transação só pode ver modificações de dados que foram confirmadas antes do início da transação. As modificações de dados feitas por outras transações após o início da transação atual não são visíveis às instruções sendo executadas na transação atual. O efeito é como se as instruções em uma transação vissem um instantâneo dos dados confirmados no estado em que se encontravam no início da transação.

- **Instantâneo de leitura confirmada**: especifica que cada instrução executada dentro de uma transação veja um instantâneo transacionalmente consistente dos dados como existia no início da instrução. As modificações de dados feitas por outras transações após o início da instrução não são visíveis à instrução. Isso é contrário ao isolamento de leitura confirmada, em que a instrução pode ver atualizações confirmadas das transações que foram confirmadas enquanto a instrução está sendo executada.

Bloqueio

O bloqueio é o principal mecanismo usado para impor a semântica dos níveis de isolamento. Todas as atualizações adquirem bloqueios exclusivos suficientes mantidos pela duração da transação para evitar que atualizações conflitantes ocorram. Bloqueios compartilhados são mantidos por várias durações a fim de fornecer os diferentes níveis de isolamento SQL para consultas.

O SQL Server fornece bloqueio de multigranularidade, que permite que diferentes tipos de recursos sejam bloqueados por uma transação (veja a Figura 29.4, na qual os recursos são listados em ordem de granularidade). Para minimizar o custo do bloqueio, o SQL Server bloqueia recursos automaticamente em uma granularidade apropriada para a tarefa. Bloquear em uma granularidade menor, como em linhas, aumenta a concorrência, mas gera um overhead maior porque mais bloqueios precisam ser mantidos se muitas linhas forem bloqueadas.

Os modos de bloqueio fundamentais do SQL Server são compartilhado (S), atualização (U) e exclusivo (X); os bloqueios de intenção também são aceitos para bloqueio de multigranularidade. Os bloqueios de atualização são usados para evitar uma forma comum de impasse que ocorre quando várias sessões estão lendo, bloqueando e potencialmente atualizando recursos mais tarde. Modos de bloqueio adicionais – chamados bloqueios de faixa de chave – são usados apenas no nível de isolamento serializável para bloquear a faixa entre duas linhas em um índice.

Bloqueio dinâmico

O bloqueio de granularidade fina pode melhorar a concorrência, ao custo de ciclos de CPU extras e da memória, para adquirir e manter muitos bloqueios. Para muitas consultas, uma granularidade de bloqueio mais fina fornece melhor desempenho sem nenhuma (ou com mínima) perda de concorrência. Os sistemas de banco de dados tradicionalmente exigem indicações e opções de tabela para que as aplicações especifiquem a granularidade de bloqueio. Além disso, existem parâmetros de configuração (normalmente estáticos) para quanta memória dedicar ao gerenciador de bloqueio.

No SQL Server, a granularidade de bloqueio é otimizada automaticamente de modo a ter desempenho e concorrência ótimos para cada índice usado em uma consulta. Além disso, a memória dedicada ao gerenciador de bloqueio é ajustada dinamicamente com base no feedback das outras partes do sistema, incluindo outras aplicações na máquina.

A granularidade de bloqueio é otimizada antes da execução da consulta para cada tabela e índice usados na consulta. O processo de otimização leva em consideração o nível de isolamento (ou seja, por quanto tempo os bloqueios são mantidos), o tipo de varredura (faixa, sondagem ou tabela inteira), o número estimado de linhas a serem varridas, a seletividade (porcentagem de linhas visitadas que se qualificam para a consulta), a densidade de linha (número de linhas por página), o tipo de operação (varredura, atualização), os limites de usuário sobre a granularidade e a memória disponível do sistema.

Quando uma consulta está sendo executada, a granularidade de bloqueio é dimensionada automaticamente para o nível de tabela se o sistema adquirir significativamente mais bloqueios do que o otimizador esperava ou se a quantidade de memória disponível cair e não puder suportar o número de bloqueios necessários.

Detecção de impasse

O SQL Server detecta automaticamente impasses envolvendo bloqueios e outros recursos. Por exemplo, se a transação

Recurso	*Descrição*
RID	Identificador de linha; usado para bloquear uma única linha dentro de uma tabela
Chave	Bloqueio de linha dentro de um índice; protege as faixas de chave nas transações serializáveis
Página	Tabela ou página de índice de 8 kilobytes
Extensão	Grupo contíguo de oito páginas de dados ou páginas de índice
Tabela	Tabela inteira, incluindo todos os dados e índices
DB	Banco de dados

Figura 29.4 Recursos bloqueáveis.

A está mantendo um bloqueio na Tabela1 e está esperando que a memória se torne disponível, e a transação B tem alguma memória que não pode liberar até adquirir um bloqueio na Tabela1, as transações estarão em um impasse. Threads e buffers de comunicação também podem se envolver em impasses. Quando o SQL Server detecta um impasse, ele escolhe como vítima do impasse a transação cuja reversão seria menos onerosa, considerando a quantidade de trabalho que a transação já fez.

Uma detecção de impasse freqüente pode prejudicar o desempenho do diagramas. O SQL Server ajusta automaticamente a freqüência da detecção de impasse para a freqüência em que os impasses estão ocorrendo. Se os impasses forem raros, o algoritmo de detecção é executado a cada 5 segundos. Se forem freqüentes, eles começarão a ser verificados a cada vez que uma transação espera por um bloqueio.

Criação de versões de linha para isolamento de instantâneo

Os dois níveis de isolamento baseados em instantâneo usam a criação de versões de linha a fim de alcançar isolamento para consultas sem bloquear as consultas por trás das atualizações, e vice-versa. Sob o isolamento de instantâneo, as operações de atualização e exclusão geram versões das linhas afetadas e as armazenam em um banco de dados temporário. As versões são alvo de coleta de lixo quando não existem transações ativas que possam precisar delas. Portanto, uma consulta executada sob o isolamento de instantâneo não precisa adquirir bloqueios, podendo, em vez disso, ler as versões mais antigas de qualquer registro que é atualizado/excluído por outra transação. A criação de versões de linha também é usada para fornecer um instantâneo de uma tabela para operações de construção de índice on-line.

Recuperação e disponibilidade

O SQL Server é projetado para se recuperar de falhas de sistema e de mídia, e o sistema de recuperação pode ser dimensionado para máquinas com pools de buffers muito grandes (100 gigabytes) e milhares de unidades de disco.

Recuperação após falha

Logicamente, o log é um fluxo potencialmente infinito de registros identificados por números de seqüência de log (LSNs). Fisicamente, uma parte do fluxo é armazenada em arquivos de log. Os registros de log são salvos nos arquivos de log até que tenham sido copiados em backup e não sejam mais necessários pelo sistema para reversão ou duplicação. Os arquivos de log crescem e reduzem em tamanho para acomodar os registros que precisam ser armazenados. Arquivos de log adicionais podem ser incluídos em um banco de dados (em novos discos, por exemplo) enquanto o sistema está sendo executado sem bloquear quaisquer operações atuais, e todos os registros são tratados como se fossem um arquivo contínuo.

O sistema de recuperação de falhas do SQL Server possui muitos aspectos em comum com o algoritmo de recuperação ARIES (veja a seção "ARIES" do Capítulo 17), e algumas das principais diferenças são destacadas nesta seção.

O SQL Server possui uma opção de configuração chamada **intervalo de recuperação**, que permite que um administrador limite o tempo que o SQL Server deve levar para se recuperar após uma falha. O servidor ajusta dinamicamente a freqüência do ponto de verificação a fim de reduzir o tempo de recuperação para dentro do intervalo de recuperação. Os pontos de verificação limpam todas as páginas sujas do pool de buffers e se ajustam às capacidades do sistema de E/S e do seu workload atual para efetivamente eliminar qualquer impacto sobre as transações em execução.

Na inicialização após uma falha, o sistema inicia múltiplas threads (automaticamente dimensionadas para o número de CPUs) a fim de começar a recuperar vários bancos de dados em paralelo. A primeira fase de recuperação é uma passagem de análise no log, que constrói uma tabela de página suja e uma lista de transação ativa. A próxima fase é uma passagem de refazer começando do último ponto de verificação e refazendo todas as operações. Durante essa fase, a tabela de página suja é usada para controlar páginas de dados de leitura antecipada. A última fase é uma passagem de desfazer, em que transações incompletas são revertidas. A fase de desfazer, na verdade, é dividida em duas partes quando o SQL Server usa um esquema de recuperação em dois níveis. As transações no primeiro nível (aquelas envolvendo operações internas, como alocação de espaço e divisões de página) são revertidas primeiro, seguidas das transações do usuário. Uma vez que as transações no primeiro nível são revertidas, o banco de dados é colocado on-line e se torna disponível para que novas transações do usuário iniciem enquanto as últimas operações de reversão são realizadas. Isso é obtido fazendo com que a passagem de refazer readquira bloqueios para todas as transações incompletas do usuário que serão revertidas na fase de desfazer.

Recuperação de mídia

As capacidades de backup e restauração do SQL Server permitem a recuperação após muitas falhas, como perda ou dano nos meios de disco, erros do usuário e perda permanente de um servidor. Além disso, o backup e a restauração de bancos de dados são úteis para outras finalidades, como copiar um banco de dados de um servidor para outro e manter sistemas em espera.

O SQL Server possui três modelos de recuperação diferentes que os usuários podem escolher de cada banco de dados. Especificando um modelo de recuperação, o administrador declara o tipo de capacidades de recuperação exigidas (como restauração point-in-time e emissão de log) e os backups necessários para alcançá-las. Os backups podem ser feitos nos bancos de dados, arquivos, filegroups e log de transação. Todos os backups são difusos e completamente on-line; ou seja, eles não bloqueiam quaisquer operações DML ou DDL enquanto são executados. As restaurações também podem ser feitas on-line de modo que apenas a parte do banco de dados sendo restaurada (por exemplo, um bloco de disco danificado) seja tomada offline. As operações de backup e restauração são altamente otimizadas e limitadas apenas pela velocidade da mídia em que o backup é armazenado. O SQL Server pode efetuar backup para dispositivos de disco e de fita (até 64 em paralelo) e possui APIs de backup de alto desempenho para uso por produtos de backup de fornecedores independentes.

Espelhamento de banco de dados

O espelhamento de banco de dados envolve reproduzir imediatamente cada atualização de um banco de dados (o banco de dados principal) em uma cópia separada e completa do banco de dados (o banco de dados espelho) geralmente localizado em outra máquina. No evento de um acidente no servidor principal, ou mesmo de simples manutenção, o sistema pode se transferir automaticamente para o espelho em questão de segundos. Uma estreita correspondência entre o banco de dados principal e o espelho é obtida enviando blocos do log de transação para o espelho enquanto ele é gerado no banco de dados principal e refazendo os registros de log no espelho. No modo de segurança completa, uma transação não pode ser confirmada até que os registros de log para a transação tenham sido gravados no disco do espelho. A biblioteca de comunicação usada pelas aplicações é ciente do espelhamento e será automaticamente reconectada à máquina espelho no caso de uma recuperação após falha.

Arquitetura do sistema

Uma única instância do SQL Server é um único processo de sistema operacional que também é uma entrada nomeada para requisições de execução SQL. As aplicações interagem com o SQL Server por intermédio de várias bibliotecas do lado do cliente (como ODBC e OLE-DB) para executar SQL.

Pool de threads no servidor

A fim de minimizar a troca de contexto no servidor e controlar o grau de multiprogramação, o processo do SQL Server mantém um pool de threads que executa requisições do cliente. À medida que as requisições chegam do cliente, elas recebem a atribuição de uma thread na qual executar. A thread executa as instruções SQL emitidas pelo cliente e envia o resultado de volta para ele. Quando a requisição do usuário é completada, a thread é retornada novamente para o pool de threads. Além das requisições do usuário, o pool de threads é usado para atribuir threads para tarefas de segundo plano internas como:

- **Lazywriter**: essa thread é dedicada a garantir que uma certa quantidade do pool de buffers esteja livre e disponível todo o tempo para alocação pelo sistema. A thread também interage com o sistema operacional para determinar a quantidade ótima de memória que deve ser consumida pelo processo do SQL Server.
- **Ponto de verificação**: essa thread insere periodicamente pontos de verificação em todos os bancos de dados a fim de manter um intervalo de recuperação rápido para os bancos de dados na reinicialização do servidor.
- **Monitor de impasse**: essa thread monitora as outras threads, procurando um impasse no sistema. Ela é responsável pela detecção de impasses e também por escolher uma vítima para permitir que o sistema avance.

Quando o processador de consulta escolhe um plano paralelo para executar uma determinada consulta, ele pode alocar múltiplas threads que operam em nome da thread principal para executar a consulta. Como a família de sistemas operacionais do Windows NT fornece suporte de thread nativo, o SQL Server usa as threads do NT para sua execução. Entretanto, o SQL Server pode ser configurado para ser executado com as threads no modo de usuário além das threads de kernel em sistemas de altíssima capacidade para evitar o custo de uma troca de contexto de kernel em uma troca de thread.

Gerenciamento de memória

Existem muitos usos da memória diferentes dentro do processo do SQL Server:

- **Pool de buffers**. O maior consumidor de memória no sistema é o pool de buffers. O pool de buffers mantém um cache das páginas de banco de dados usadas mais recentemente. Ele usa o algoritmo de substituição de clock com uma política de roubo; ou seja, as páginas do buffer com atualizações não confirmadas podem ser substituídas ("roubadas"), e as páginas do buffer não são forçadas para o disco na confirmação da transação. Os buffers também obedecem ao protocolo de logging de escrita antecipada para garantir a fidelidade da recuperação de falha e de mídia.

- **Alocação de memória dinâmica.** Essa é a memória que é alocada dinamicamente para executar requisições submetidas pelo usuário.
- **Cache de plano e execução.** Esse cache armazena os planos compilados para várias consultas que tenham sido previamente executadas por usuários no sistema. Isso permite que vários usuários compartilhem o mesmo plano (poupando memória) e também economiza tempo de compilação de consulta para consultas semelhantes.
- **Grandes concessões de memória.** Para operações de consulta que consomem grandes quantidades de memória, como junção ou classificação de hash.

O SQL Server usa um esquema elaborado de gerenciamento de memória para dividir sua memória entre os vários usos descritos anteriormente. Um único gerenciador de memória controla centralmente toda a memória usada pelo SQL Server. O gerenciador de memória é responsável por particionar e redistribuir dinamicamente a memória entre os vários consumidores de memória no sistema. Ele distribui essa memória de acordo com uma análise do custo-benefício relativo da memória para qualquer uso específico. Um mecanismo de infra-estrutura LRU generalizado está disponível para todos os componentes. Essa infra-estrutura de cache rastreia não só o tempo de vida dos dados em cache, mas também os custos de CPU e E/S relativos incorridos para criá-los e colocá-los no cache. Essas informações são usadas para determinar os custos relativos dos vários dados em cache. O gerenciador de memória se concentra em retirar do cache os dados que não foram usados recentemente e não envolveram altos custos para serem colocados no cache. Como exemplo, os planos de consulta complexos que exigem segundos de tempo de CPU para serem compilados são mais propensos a permanecerem na memória do que planos triviais, dadas as freqüências de acesso equivalentes.

O gerenciador de memória interage com o sistema operacional para decidir dinamicamente quanta memória ele deve consumir da quantidade total de memória no sistema. Isso permite que o SQL Server seja bastante agressivo ao usar a memória no sistema mas ainda retorne memória para o sistema quando outros programas precisarem dela, sem causar excessivas page faults.

Segurança

O SQL Server fornece mecanismos e políticas de segurança completos para autenticação, autorização e criptografia. Dois fatores são ainda mais vitais para a segurança dos usuários: (1) a qualidade de toda a própria base de código e (2) a capacidade de os usuários determinarem se protegeram o sistema adequadamente.

A qualidade da base de código é melhorada fazendo todos os desenvolvedores e testadores do produto passarem por um treinamento de segurança. Sempre que possível, o SQL Server utiliza os recursos de segurança básicos do sistema operacional em vez de implementar os seus próprios recursos. Além disso, diversas ferramentas internas são utilizadas para analisar a base de código, procurando possíveis falhas de segurança.

Vários recursos são fornecidos para ajudar os usuários a protegerem o sistema corretamente. Um deles é uma política fundamental chamada "off-by-default" (desativado por padrão), em que muitos componentes pouco usados ou os que exigem cuidado extra para segurança são completamente desativados por padrão. Outro recurso é um "analisador de melhores práticas", que avisa os usuários sobre configurações do sistema que poderiam levar a uma vulnerabilidade de segurança.

Acesso a dados

O SQL Server aceita as seguintes interfaces de programa de aplicação (APIs) para construir aplicações com intensa utilização de dados:

- **ODBC.** Essa é uma implementação da Microsoft da interface de nível de chamada (CLI) do padrão SQL:1999. Ela inclui modelos de objeto – Remote Data Objects (RDOs) e Data Access Objects (DAOs) – que facilitam a programação de aplicações de banco de dados de múltiplas camadas através de linguagens de programação como Visual Basic.
- **OLE-DB.** Essa é uma API de baixo nível orientada a sistemas, projetada para os programadores construírem componentes de banco de dados. A interface é arquitetada de acordo com o Microsoft Component Object Model (COM) e permite a encapsulação dos serviços de banco de dados de baixo nível, como provedores de conjunto de linhas, provedores ISAM e mecanismos de busca. A OLE-DB é usada dentro do SQL Server a fim de integrar o processador de consulta relacional e o mecanismo de armazenamento e permitir a duplicação e o acesso distribuído para a SQL e outras origens de dados externas. Como a ODBC, a OLE-DB inclui um modelo de objeto de nível superior chamado ActiveX Data Objects (ADO) para facilitar a programação de aplicações de banco de dados pelo Visual Basic.
- **ADO.NET.** Essa é uma nova API projetada para aplicações escritas em linguagens .NET, como C# e Visual Basic.NET. Essa interface simplifica alguns padrões de acesso a dados comuns aceitos pela ODBC e OLE-DB. Além disso, ela fornece um novo modelo de *conjunto de dados* para permitir que dados desconectados e sem in-

formação de estado acessem aplicações.

- **DB-Lib**. A DB-Library para a API C que foi desenvolvida especificamente para ser usada com versões antigas do SQL Server que sejam anteriores ao padrão SQL-92.
- **HTTP/SOAP**. As aplicações podem usar requisições HTTP/SOAP para chamar consultas e procedimentos do SQL Server. As aplicações podem usar URLs que especificam raízes virtuais do Internet Information Server (IIS) que referenciam uma instância do SQL Server. O URL pode conter uma consulta XPath, uma instrução Transact-SQL ou um modelo XML.

Processamento de consulta heterogênea distribuída

A capacidade de consulta heterogênea distribuída do SQL Server permite consultas e atualizações transacionais em várias origens relacionais e não relacionais através de provedores de dados OLE-DB sendo executados em um ou mais computadores. O SQL Server aceita dois métodos para referenciar origens de dados OLE-DB heterogêneas em instruções Transact-SQL. O método linked-server-names (nomes de servidor interligados) usa procedimentos armazenados no sistema para associar um nome de servidor a uma origem de dados OLE-DB. Os objetos nesses servidores interligados podem ser referenciados nas instruções Transact-SQL usando a convenção de nome de quatro partes descrita a seguir. Por exemplo, se um nome de servidor interligado *DeptSQLSrvr* for definido em outra cópia do SQL Server, a seguinte instrução referencia uma tabela nesse servidor:

```
select *
from DeptSQLSrvr.Northwind.dbo.Funcionários
```

Uma origem de dados OLE-DB é registrada no SQL Server como um servidor interligado. Uma vez definido um servidor interligado, seus dados podem ser acessados usando o nome de quatro partes

```
<servidor-interligado>.<catálogo>.<esquema>.<objeto>
```

O exemplo a seguir estabelece um servidor interligado para um servidor Oracle através de um provedor OLE-DB para Oracle:

```
exec sp_addlinkedserver OraSvr, 'Oracle 7.3',
    'MSDAORA', 'OracleServer'
```

Uma consulta a esse servidor interligado é expressa como:

```
select *
from OraSvr.CORP.ADMIN.VENDAS
```

Além disso, o SQL Server aceita funções parametrizadas com valor de tabela, chamadas **openrowset** e **openquery**, que permitem enviar consultas não interpretadas a um provedor ou servidor interligado, respectivamente, no dialeto aceito pelo provedor. A seguinte consulta combina informações armazenadas em um servidor Oracle e um servidor de índice Microsoft. Ela lista todos os documentos e seus autores contendo as palavras Data e Access, ordenados pelo departamento e nome do autor.

```
select e.dept, f.DocAuthor, f.FileName
from OraSvr.Corp.Admin.Employee e,
openquery(EmpFiles,
    'select DocAuthor, FileName
    from scope("c:\EmpDocs")
    where contains(" "Data" near( ) "Access" ")>0') as f
where e.name = f.DocAuthor
order by e.dept, f.DocAuthor
```

O mecanismo relacional usa as interfaces OLE-DB para abrir os conjuntos de linhas nos servidores interligados, buscar as linhas e gerenciar transações. Para cada origem de dados OLE-DB acessada como um servidor interligado, um provedor OLE-DB precisa estar presente no servidor executando o SQL Server. O conjunto das operações Transact-SQL que podem ser usadas em uma origem de dados OLE-DB específica depende das capacidades do provedor OLE-DB. Sempre que for compensador, o SQL Server envia operações relacionais, como junções, restrições, projeções, classificações e operações group by para a origem de dados OLE-DB. O SQL Server usa o Microsoft Distributed Transaction Coordinator e as interfaces de transação OLE-DB do provedor para assegurar a atomicidade das transações que abrangem múltiplas origens de dados.

Duplicação

A duplicação do SQL Server é um conjunto de tecnologias para copiar e distribuir dados e objetos de um banco de dados para outro, controlar alterações e sincronizar entre bancos de dados, de modo a manter a consistência. As últimas versões da duplicação do SQL Server também fornecem duplicação em série da maioria das mudanças de esquema de banco de dados sem exigir qualquer interrupção ou reconfiguração.

Os dados normalmente são duplicados para aumentar sua disponibilidade. A duplicação pode coletar dados corporativos de locais geograficamente dispersos para fins de relatório e disseminar os dados para usuários remotos em

uma rede local ou usuários móveis em conexões discadas ou da Internet. A duplicação do Microsoft SQL Server também melhora o desempenho da aplicação dimensionando para o desempenho de leitura total aprimorada entre cópias, como é comum ao fornecer serviços de cache de dados de camada intermediária para sites.

Modelo de duplicação

O SQL Server introduziu a metáfora da *assinatura de publicações* na duplicação de banco de dados e estende essa metáfora do setor de publicações para todas as suas ferramentas de administração e monitoramento de duplicação.

O **editor** é um servidor que torna os dados disponíveis para duplicação em outros servidores. O editor pode ter uma ou mais publicações, cada uma representando um conjunto de dados e objetos de banco de dados logicamente relacionados. Os diferentes objetos dentro de uma publicação, incluindo tabelas, procedimentos armazenados, funções definidas pelo usuário, views, views materializadas e mais, são chamados **artigos**. A adição de um artigo em uma publicação permite a personalização extensiva da maneira como o objeto é duplicado, por exemplo, restrições sobre quais usuários podem assinar para receberem seus dados e como os dados devem ser filtrados com base em uma projeção ou seleção de uma tabela, por um filtro "horizontal" ou "vertical", respectivamente.

Os **assinantes** são servidores que recebem dados duplicados de um editor. Os assinantes podem assinar convenientemente apenas as publicações que necessitam de um ou mais editores, seja qual for o número ou o tipo de opções de duplicação que cada um implementa. Dependendo do tipo de opções de duplicação escolhido, o assinante pode ser usado como uma cópia de leitura ou pode fazer alterações de dados que são automaticamente propagadas de volta ao editor e, subsequentemente, a todas as outras cópias. Os assinantes também podem republicar os dados para os quais assinaram, aceitando uma topologia de duplicação tão flexível quanto a empresa exige.

O **distribuidor** é um servidor que desempenha diferentes funções, dependendo das opções de duplicação escolhidas. No mínimo, ele é usado como um depósito para informações de estado de histórico e erro. Em outros casos, ele é usado adicionalmente como uma fila store-and-forward (armazenar e repassar) intermediária para dimensionar a entrega do payload duplicado para todos os assinantes.

Opções de duplicação

A duplicação do Microsoft SQL Server oferece um amplo espectro de opções de tecnologia. Para decidir sobre as opções de duplicação apropriadas, um projetista de banco de dados precisa determinar as necessidades da aplicação no que se refere à operação autônoma do local envolvido e o grau de consistência transacional exigido.

A **duplicação de instantâneo** copia e distribui dados e objetos de banco de dados exatamente conforme aparecem em um dado momento no tempo. A duplicação de instantâneo não exige monitoramento de mudanças contínuo, já que as alterações não são propagadas de modo incremental aos assinantes. Os assinantes recebem uma atualização completa do conjunto de dados definido pela publicação de maneira periódica. As opções disponíveis com a duplicação de instantâneo podem filtrar dados publicados e podem permitir que os assinantes modifiquem dados duplicados e propaguem essas mudanças de volta para o editor. Esse tipo de duplicação é mais adequado para pequenos tamanhos de dados e quando as atualizações normalmente afetam os dados o bastante para que a duplicação de uma atualização completa dos dados seja eficiente.

Com a **duplicação transacional**, o editor propaga um instantâneo inicial dos dados para os assinantes e, depois, repassa as modificações de dados incrementais aos assinantes como transações e comandos separados. O monitoramento de mudança incremental ocorre dentro do mecanismo central do SQL Server, que marca as transações afetando objetos duplicados no log de transação do banco de dados da publicação. Um processo de duplicação chamado **agente leitor do log** lê essas transações do log de transação do banco de dados, aplica um filtro opcional e as armazena no banco de dados da distribuição, que age como a fila segura aceitando o mecanismo store-and-forward da duplicação transacional. (Filas seguras são o mesmo que filas duráveis, descritas na seção "Monitores de processamento de transação" do Capítulo 25.) Em seguida, outro processo de duplicação, chamado **agente de distribuição**, encaminha as mudanças para cada assinante. Como a duplicação de instantâneo, a duplicação transacional oferece aos assinantes a opção de fazer atualizações que podem (1) usar confirmação de duas fases para refletir essas mudanças consistentemente no editor e no assinante ou (2) podem colocar as mudanças em uma fila no assinante para recuperação assíncrona por um processo de duplicação que, posteriormente, propaga a mudança para o editor. Esse tipo de duplicação é apropriado quando estados intermediários entre múltiplas atualizações precisam ser preservados.

A **duplicação de mesclagem** permite que cada cópia na empresa trabalhe com uma autonomia total, seja on-line ou offline. O sistema monitora os metadados sobre as mudanças nos objetos publicados nos editores e assinantes em cada banco de dados duplicado, e o agente de duplicação mescla essas modificações de dados durante a sincronização entre pares duplicados e garante a convergência dos dados por intermédio da detecção e da resolução de conflitos

automática. Diversas opções de política de resolução de conflitos estão embutidas no agente de duplicação usado no processo de sincronização, e uma resolução de conflitos personalizada pode ser escrita usando procedimentos armazenados ou usando uma interface de **modelo de objeto componente (COM)**. Esse tipo de duplicação não duplica todos os estados intermediários, mas apenas o estado atual dos dados no momento da sincronização. Ele é adequado quando as cópias exigem a capacidade de fazer atualizações autônomas quando não conectadas a uma rede.

Programação do servidor na .NET

O SQL Server aceita a hospedagem do Common Language Runtime (CLR) .NET dentro do processo do SQL Server para permitir que programadores de banco de dados escrevam lógica empresarial, como funções, procedimentos armazenados, acionadores, tipos de dados e agregados. A capacidade de executar código de aplicação dentro do banco de dados acrescenta flexibilidade ao projeto das arquiteturas de aplicação, que exige que a lógica empresarial seja executada próximo aos dados e não disponha do custo de enviar dados a um processo de camada intermediária para realizar computação fora do banco de dados.

O Common Language Runtime (CLR) .NET é um ambiente de tempo de execução com uma linguagem intermediária fortemente tipificada que executa várias linguagens de programação modernas, como C#, Visual Basic, C++, COBOL e J++, entre outras, possui coleta de lixo de memória, threading preemptivo, serviços de metadados (reflexão de tipo), capacidade de verificação de código e segurança de acesso de código. O runtime usa metadados para localizar e carregar classes, dispor instâncias na memória, resolver chamadas de método, gerar código nativo, impor segurança e definir limites de contexto de tempo de execução.

O código de aplicação é utilizado dentro do banco de dados usando assemblies, que são unidades de empacotamento, utilização e geração de versão do código de aplicação na .NET. A utilização do código de aplicação dentro do banco de dados fornece uma maneira uniforme de administrar, copiar e restaurar aplicações de banco de dados completas (código e dados). Uma vez que um assembly é registrado dentro do banco de dados, os usuários podem expor pontos de entrada dentro do assembly através de instruções DDL SQL, que podem agir como funções de tabela ou escalares, procedimentos, acionadores, tipos e agregados, usando contratos de extensibilidade bem definidos, impostos durante a execução dessas instruções DDL. Os procedimentos armazenados, acionadores e funções normalmente precisam executar consultas e atualizações SQL. Isso é conseguido por meio de um componente que implementa a API de acesso a dados ADO.NET para uso dentro do processo de banco de dados.

Conceitos básicos da .NET

Na estrutura .NET, um programador escreve um código de programa em uma linguagem de programação de alto nível que implementa uma classe definindo sua estrutura (por exemplo, os campos ou propriedades da classe) e seus métodos. Alguns desses métodos podem ser funções estáticas. A compilação do programa produz um arquivo, chamado um *assembly*, contendo o código compilado na *Microsoft Intermediate Language (MSIL)*, e um *manifesto* contendo todas as referências aos assemblies dependentes. O manifesto é uma parte integrante de cada assembly que torna o assembly autodescritivo. O manifesto do assembly contém os metadados do assembly, que descrevem todas as estruturas, campos, propriedades, classes, relacionamentos de herança, funções e métodos definidos no programa. O manifesto estabelece a identidade do assembly, especifica os arquivos que compõem a implementação do assembly, especifica os tipos e recursos que compõem o assembly, relaciona as dependências de outros assemblies em tempo de compilação e determina o conjunto de permissões necessárias para que o assembly seja executado corretamente. Essas informações são usadas em tempo de execução para resolver referências, impor política de associação de versão e validar a integridade de assemblies carregados. A estrutura .NET possui suporte a um mecanismo out-of-band chamado *atributos personalizados* para anotar classes, propriedades, funções e métodos com informações ou aspectos adicionais que a aplicação pode querer capturar nos metadados. Todos os compiladores .NET consomem essas anotações sem interpretação e as armazena nos metadados do assembly. Todas essas anotações podem ser examinadas da mesma maneira que quaisquer outros metadados usando um conjunto comum de APIs de reflexão. *Código gerenciado* se refere à execução da MSIL no CLR em vez de diretamente pelo sistema operacional. As aplicações de código gerenciado ganham serviços de runtime de linguagem comum, como coleta de lixo automática, verificação de tipo em tempo de execução e suporte a segurança. Esses serviços ajudam a fornecer um comportamento independente de plataforma e linguagem das aplicações de código gerenciado. Em tempo de execução, um compilador just-in-time (JIT) traduz a MSIL para código nativo (por exemplo, código do Intel X86). Durante essa tradução, o código precisa passar por um processo de verificação que examina a MSIL e os metadados para descobrir se o código pode ser especificado como seguro para tipos.

Hospedagem CLR SQL

O SQL Server e o CLR são dois runtimes diferentes com variados modelos internos para threading, escalonamento e gerenciamento de memória. O SQL Server aceita um mode-

o de threading cooperativo e não preemptivo no qual as hreads DBMS liberam a execução voluntária e periodicamente ou quando estão esperando bloqueios ou E/S, enquanto o CLR aceita um modelo de threading preemptivo. e o código usuário sendo executado dentro do DBMS puer chamar diretamente as primitivas de threading do sistema operacional, então, ele não se integrará bem com o escanador de tarefa do SQL Server e poderá comprometer a apacidade de dimensionamento do sistema. O CLR não istingue entre memória virtual e memória física, enquanto SQL Server gerencia diretamente a memória física e é necessário para usar a memória física dentro de limites configuráveis.

Os diversos modelos de threading, escalonamento e genciamento de memória apresentam um problema de interação para um DBMS que é dimensionado para aceitar miares de sessões concorrentes de usuário. O SQL Server resolve esse problema se tornando o sistema operacional para CLR quando este está hospedado dentro do processo do QL Server. O CLR chama primitivas de baixo nível implementadas pelo SQL Server para threading, escalonamento, ncronização e gerenciamento de memória (veja a Figura 9.5). Esse método oferece as vantagens de dimensionabilidade e confiabilidade apresentadas a seguir.

Threading comum, escalonamento e sincronização. O LR chama as APIs do SQL Server para criar threads tanto ara executar código do usuário quanto para seu próprio so interno, como o coletor de lixo e o thread finalizador de asse. Para sincronizar entre múltiplas threads, o CLR chaa os objetos de sincronização do SQL Server. Isso permite ue o escalonador do SQL Server programe outras tarefas uando um thread está esperando em um objeto de sincroização. Por exemplo, quando o CLR inicia a coleta de lixo, todas as suas threads esperam a coleta de lixo terminar. Como as threads CLR e os objetos de sincronização em que eles estão esperando são conhecidos do escalonador do SQL Server, ele pode programar threads que estejam executando outras tarefas de banco de dados não envolvendo o CLR. Além do mais, isso permite que o SQL Server detecte impasses que envolvam bloqueios tomados por objetos de sincronização do CLR e empregue técnicas tradicionais para remoção do impasse. O escalonador do SQL Server tem a capacidade de detectar e interromper threads que não produziram por um período de tempo significativo. A capacidade de enganchar threads CLR no SQL Server implica que o escalonador do SQL Server pode identificar threads fugitivas sendo executadas no CLR e gerenciar sua prioridade, de modo que não consumam recursos de CPU significativos, afetando, assim, a vazão do sistema. Essas threads fugitivas são suspensas e enviadas de volta para a fila. As ofensoras reincidentes não recebem fatias de tempo que sejam injustas para outras operárias em execução. Se uma ofensora levou 50 vezes a cota permitida, ela é punida por 50 "rodadas" antes que possa ser executada novamente, pois o escalonador não pode saber quando uma computação é longa e inválida ou longa e legítima.

Gerenciamento de memória comum. O CLR chama primitivas do SQL Server para alocar e desalocar sua memória. Como a memória usada pelo CLR é levada em conta para o uso de memória total do sistema, o SQL Server pode ficar dentro de seus limites de memória configurados e garantir que o CLR e o SQL Server não estejam competindo entre si pela memória. Além disso, o SQL Server pode rejeitar as requisições de memória do CLR quando o sistema é restrito e pedir que o CLR reduza seu uso de memória quando outras tarefas precisarem da memória.

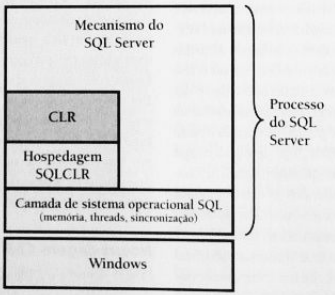


Figura 29.5 Integração do CLR com os serviços de sistema operacional do SQL Server.

Contratos de extensibilidade

Todo código gerenciado pelo usuário sendo executado dentro do processo do SQL Server interage com os componentes DBMS como uma extensão. As extensões atuais incluem funções escalares, funções de tabela, procedimentos, acionadores, tipos escalares e agregados escalares. Para cada extensão, existe um contrato mútuo definindo as propriedades ou serviços que o código do usuário precisa implementar para agir como uma dessas extensões, bem como os serviços que a extensão pode esperar do DBMS quando o código gerenciado é chamado. O CLR SQL extrai as informações de classe e de atributos personalizados armazenadas no metadados do assembly para garantir que o código do usuário implemente esses contratos de extensibilidade. Todos os assemblies do usuário são armazenados dentro do banco de dados. Todos os metadados relacionais e do assembly são processados dentro do mecanismo SQL através de um conjunto uniforme de interfaces e estruturas de dados. Quando instruções DDL registrando uma determinada função de extensão, tipo ou agregado são processadas, o sistema garante que o código do usuário implemente o contrato apropriado analisando seus metadados do assembly. Se o contrato for implementado, então, a instrução DDL é executada; caso contrário, ela falha. As próximas subseções descrevem os aspectos fundamentais dos contratos específicos atualmente em vigor pelo SQL Server.

Rotinas

Classificamos funções escalares, procedimentos e acionadores genericamente como rotinas. Implementadas como métodos de classe estáticos, as rotinas podem especificar as seguintes propriedades através de atributos personalizados.

- **IsPrecise**. Se essa propriedade booleana for **false**, então, ela indica que o corpo da rotina envolve cálculos imprecisos, como operações de ponto flutuante. As expressões envolvendo funções imprecisas não podem ser indexadas.
- **UserDataAccess**. Se o valor dessa propriedade for **read**, então, a rotina lê tabelas de dados do usuário. Caso contrário, o valor da propriedade é None, indicando que a rotina não acessa dados. As consultas que não acessam quaisquer tabelas do usuário (direta ou indiretamente através de views e funções) não são consideradas como tendo acesso aos dados do usuário.
- **SystemDataAccess**. Se o valor dessa propriedade for **read**, então, a rotina lê catálogos do sistema ou tabelas de sistema virtuais.
- **IsDeterministic**. Se essa propriedade for **true**, então, a rotina é considerada como produzindo o mesmo valor de saída, considerando os mesmos valores de entrada, estado do banco de dados local e contexto de execução.
- **IsSystemVerified**. Indica se as propriedades determinismo e precisão podem ser verificadas ou impostas pelo SQL Server (por exemplo, funções Transact-SQL internas) ou se elas são conforme especificadas pelo usuário (por exemplo, funções CLR).
- **HasExternalAccess**. Se o valor dessa propriedade for **true**, então, a rotina acessa recursos fora do SQL Server, como arquivos, rede, acesso a Web e registro.

Funções de tabela

Uma classe implementando uma função com valor de tabela precisa implementar uma interface IEnumerable para permitir repetição através das linhas retornadas pela função, um método para descrever o esquema da tabela retornada (por exemplo, colunas e tipos), um método para descrever que colunas podem ser chaves únicas e um método para inserir linhas na tabela.

Tipos

As classes implementando tipos definidos pelo usuário são anotadas com um atributo SqlUserDefinedType( ), que especifica as seguintes propriedades:

- **Format**. O SQL Server aceita três formatos de armazenamento: nativo, definido pelo usuário e serialização .NET.
- **MaxByteSize**. Esse é o tamanho máximo, em bytes, da representação binária serializada das instâncias de tipo.
- **IsFixedLength**. Essa é uma propriedade booleana que especifica se as instâncias do tipo possuem tamanho fixo ou variável.
- **IsByteOrdered**. Essa é uma propriedade booleana que especifica se a representação binária serializada das instâncias de tipo é ordenada por binário. Quando essa propriedade é **true**, o sistema pode realizar comparações diretamente com relação a essa representação sem a necessidade de instanciar instâncias de tipo como objetos.
- **Nulidade**. Todos os UDTs em nosso sistema precisam ser capazes de armazenar o valor NULL aceitando a interface INullable contendo o método booleano IsNull.
- **Conversões de tipo**. Todos os UDTs precisam implementar conversões de e para strings de caractere através dos métodos ToString e Parse.

Agregados

Além de aceitar o contrato para tipos, os agregados definidos pelo usuário precisam implementar quatro métodos exigidos pelo mecanismo de execução de consulta para ini-

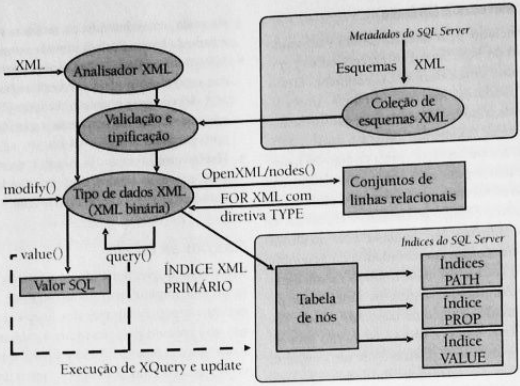


Figura 29.6 Resumo arquitetônico do suporte a XML nativo no SQL Server.

cializar o cálculo de uma instância agregada, mesclar cálcu-os parciais do agregado e recuperar o resultado agregado final. Os agregados também podem declarar propriedades adicionais, através de atributos personalizados, em sua de-finição de classe; essas propriedades são usadas pelo otimi-zador de consulta a fim de criar planos alternativos para o cálculo agregado.

- **IsInvariantToDuplicates**. Se essa propriedade for **true**, então, o cálculo distribuindo os dados para o agregado pode ser modificado descartando ou introduzindo no-vas operações de remoção de duplicação.
- **IsInvariantToNulls**. Se essa propriedade for **true**, então, linhas NULL podem ser descartadas da entrada em al-guns casos, mas é preciso ter cuidado para não descartar grupos inteiros.
- **IsInvariantToOrder**. Se essa propriedade for **true**, en-tão, o processador de consulta pode ignorar cláusulas **order by** e explorar planos que evitem a necessidade de classificar os dados.

Suporte a XML no SQL Server 2005

Os sistemas de banco de dados relacionais incorporaram a XML de muitas maneiras diferentes ao longo dos últimos dois anos. O suporte a XML da primeira geração se con-centrava principalmente em exportar dados relacionais como XML ("publicação XML") e em importar dados rela-cionais na forma de marcação XML de volta para uma re-presentação relacional ("fragmentação XML"). O princi-pal cenário de uso aceito por esses sistemas é a troca de in-formações em contextos em que a XML é usada como o "formato de transporte" e os esquemas relacional e XML normalmente são predefinidos de maneira independente um do outro. Para cobrir esse cenário, o Microsoft SQL Server 2005 fornece e estende a extensiva funcionalidade introduzida inicialmente no SQL Server 2000, como o agregador de conjunto de linhas de publicação **for xml**, o provedor de conjunto de linhas OpenXML e a tecnologia de view XML baseada em esquemas anotados. Veja as no-tas bibliográficas para obter referências com mais infor-mações sobre recursos XML no SQL Server 2000.

A fragmentação de dados XML em um esquema relacio-nal pode ser muito difícil ou ineficiente para armazenar dados semi-estruturados cuja estrutura pode variar com o tempo e para armazenar documentos. A fim de aceitar es-sas aplicações, o SQL Server 2005 acrescenta suporte a XML nativo baseado no tipo de dados **xml** do padrão SQL:2003. A Figura 29.6 fornece um diagrama arquitetô-nico de alto nível do suporte a XML nativo do SQL Server no banco de dados. Ele consiste na capacidade de armaze-nar XML nativamente, de restringir e tipificar os dados XML armazenados com coleções de esquemas XML e de consultar e atualizar os dados XML. Para fornecer execu-ções de consulta eficientes, vários tipos de índices especí-ficos da XML são fornecidos. Finalmente, o suporte a XML nativo também se integra à "fragmentação" e "publi-cação" de e para dados relacionais.

Armazenando e organizando nativamente a XML

O tipo de dados **xml** pode armazenar documentos XML e fragmentos de conteúdo (múltiplo texto ou nós de elemento no alto) e é definido como a base do modelo de dados XQuery 1.0/XPath 2.0. O tipo de dados pode ser usado para parâmetros de procedimentos armazenados, variáveis e como um tipo de coluna. Por exemplo, a seguinte instrução SQL cria uma tabela em que uma das colunas é do tipo XML:

```
create table RelatViagem(id int,
    dataviagem datetime,
    Relatório xml)
```

Um sistema de banco de dados pode escolher armazenar uma instância do tipo **xml** em qualquer um entre vários formatos, como um objeto grande binário ou caractere, decomposto em tabelas ou uma mistura desses. O SQL Server 2005 armazena dados do tipo **xml** em um formato binário interno como um **blob** e fornece mecanismos de indexação para executar consultas. O formato binário interno fornece uma eficiente recuperação e reconstrução do documento XML original, além de alguma economia de espaço (em média, 20%). Os índices aceitam um mecanismo de consulta eficiente que pode utilizar o mecanismo e o otimizador de consulta relacionais; mais detalhes são fornecidos posteriormente, na seção "Execução de expressões XQuery".

Validando e tipificando um tipo de dados XML

O SQL Server oferece a capacidade de restringir um tipo de dados XML com uma coleção de esquemas XML. O SQL Server 2005 fornece um conceito de metadados de banco de dados chamado **coleção de esquemas XML**, que associa um identificador SQL a uma coleção de componentes de esquema de um ou vários espaços de nomes de destino. Por exemplo, a expressão

```
create xml schema collection S1 as @s
```

cria uma coleção de esquemas XML do SQL Server com o nome *S1* que consiste nos esquemas XML contidos pela variável SQL @s. Cada uma dessas coleções contém todas as informações necessárias para realizar validação e tipificação e é armazenada nos metadados do esquema de banco de dados.

O SQL Server 2005, então, permite associar uma coleção de esquemas a um tipo de dados XML e especificar se o tipo pode conter apenas um documento (por padrão, ele pode conter fragmentos de documento). O exemplo a seguir mostra uma definição de tabela que restringe as instâncias na coluna XML *Relatório* a um documento bem formado (isto é, ele pode ter apenas um único elemento de nível superior) que é válido de acordo com uma coleção de esquemas *EsquemaRelatório* no mesmo banco de dados:

```
create table RelatViagem(id int,
    dataviagem datetime,
    Relatório xml(document EsquemaRelatório))
```

Descartar a palavra-chave **document** permitiria que conteúdo de múltiplos fragmentos fosse armazenado.

As coleções de esquemas também podem ser usadas para validar XML não previamente tipificada durante a execução realizando um cast de tipo:

```
cast (@x as xml(S1))
```

Consultando e atualizando o tipo de dados XML

O SQL Server 2005 fornece várias capacidades de consulta e modificação baseadas no XQuery sobre o tipo de dados XML. Essas capacidades são aceitas usando métodos definidos no tipo de dados **xml**. Alguns desses métodos são descritos no restante desta seção.

Cada método toma uma literal de string como a string de consulta e potencialmente outros argumentos. O tipo de dados XML (no qual o método é aplicado) fornece o item de contexto para as expressões de caminho e preenche as definições de esquema no escopo, com todas as informações de tipo fornecidas pela coleção de esquemas XML associada (se nenhuma coleção for fornecida, os dados XML são considerados não tipificados). A implementação XQuery do SQL Server 2005 é estaticamente tipificada, aceitando, portanto, detecção precoce de erros de digitação de expressão de caminho, erros de tipo e inadequação de cardinalidade, bem como algumas otimizações adicionais.

O método *query* toma uma expressão XQuery e retorna uma instância de tipo de dados XML (que pode, então, ser convertida em uma coleção de esquemas se os dados precisarem ser tipificados). Na terminologia da especificação XQuery, definimos o modo de construção para "strip". O exemplo a seguir mostra uma expressão XQuery simples que resume um elemento `Customer` complexo em um documento de relatório de viagem que contém, entre outras informações, um nome, um atributo ID e informações de chefia de vendas que estão contidas nas notas marcadas do relatório de viagem real. O resumo mostra o nome e chefias de venda para elementos Cliente que possuem chefias de venda.

```
select Relatório.consulta('
    declare namespace c = "urn:exemplo/cliente";
```

```
    for $cli in /c:doc/c:cliente
    where $cli/c:notas//c:chefvendas
    return
        <id_cliente="$cli/@id"> {
        $cli/c:nome,
        $cli/c:notas//c:chefvendas
    }</cliente>')
from RelatóriosViagem
```

Essa consulta XQuery é executada no valor XML armazenado no atributo *doc* de cada linha da tabela *Relatórios Viagem*. Cada linha no resultado da consulta SQL contém o resultado de executar a consulta XQuery nos dados em uma linha de entrada.

O método *valor* toma uma expressão XQuery e um nome de tipo SQL, extrai um único valor atômico do resultado da expressão XQuery e converte sua forma lógica no tipo SQL especificado. Se a expressão XQuery resultar em um nó, o valor tipificado do nó será implicitamente extraído como o valor atômico a ser convertido no tipo SQL (na terminologia XQuery, o nó será "atomizado"; o resultado é convertido em SQL). Note que o método *valor* realiza uma verificação de tipo estático de que no máximo um valor esteja sendo retornado. Como o tipo estático de uma expressão de caminho normalmente pode inferir um tipo estático mais amplo, ainda que a semântica retorne apenas um único valor, recomendamos usar predicado posicional para recuperar no máximo um valor. O exemplo a seguir mostra uma expressão XQuery simples que conta os elementos de chefia de vendas em cada instância de tipo de dados XML e retorna como um valor inteiro SQL:

```
select Relatório.valor(
    'declare namespace c = "urn:exemplo/cliente";
    count(/c:doc/c:cliente//c:chefvendas)', 'int')
from RelatóriosViagem
```

O método *exist* toma uma expressão XQuery e retorna 1 se a expressão produzir um resultado não vazio e 0, caso contrário. A seguinte expressão recupera cada linha da tabela *RelatóriosViagem*, em que o documento contém pelo menos um cliente com uma chefia de vendas:

```
select Relatório
from RelatóriosViagem
where 1 = Relatório.exist('declare namespace c =
"urn:exemplo/cliente";
                /c:doc/c:cliente//c:chefvendas)'
```

Até agora, as expressões sempre mapeiam de uma instância de tipo de dados XML para um valor resultado por linha adicional. Algumas vezes, no entanto, você deseja dividir uma instância XML em várias subárvores, em que cada subárvore está sozinha em uma linha, para mais processamento relacional e XQuery. Essa funcionalidade é fornecida pelo método *nodes*, que toma uma expressão XQuery e gera uma tabela contendo linhas de coluna única, com uma linha por nó que a expressão retorna. Cada linha contém uma referência a um dos nós. Como o tipo resultante é um tipo de referência que não existe no SQL Server fora do contexto de uma única consulta, um dos métodos de consulta precisa ser aplicado para materializar o resultado. O método de consulta é aplicado como qualquer outro tipo de dados XML, com a diferença de que o item de contexto para as expressões de caminho não está no documento raiz do tipo de dados XML, mas no nó referenciado. O exemplo a seguir extrai, para cada pedido de cliente na coluna XML, uma linha que contém a representação XML do seu cliente, o nome do cliente, o ID do pedido e o ID do documento que contém o pedido:

```
select N1.cliente.consulta('.') as Cliente,
        N1.cliente.value(
          'declare namespace c = "urn:exemplo/cliente";
          c:name[1]','nvarchar(20)') as NomeCliente,
        N2."pedido".value('@id','int') as IDPedido,
        N1.cliente.value('../@id','nvarchar(5)') as IDDoc
from RelatóriosViagem cross apply
        RelatóriosViagem.Relatório.nodes(
          'declare namespace c = "urn:exemplo/cliente";
          /c:doc/c:cliente') as N1(cliente)
        cross apply N1.cliente.nodes(
          'declare namespace c = "urn:exemplo/cliente";
          ./c:pedido') as N2("pedido")
```

Observe que isso é semelhante à funcionalidade OpenXML fornecida pelo SQL Server 2000 e 2005, com a diferença de que a expressão do método *nodes* é integrada no processamento XQuery.

Para acessar os dados SQL dentro de uma expressão XQuery, o SQL Server 2005 oferece duas funções chamadas *sql:variable*(*$varname as xs:string*) e *sql:column*(*$colname as xs:string*). Cada função toma uma literal de string constante para se referir à variável SQL ou ao valor de coluna correlato.

Finalmente, o método *modify* fornece um mecanismo para mudar um valor XML no nível de subárvore. O SQL Server 2005 permite inserir novas subárvores em locais específicos dentro de uma árvore, mudando o valor de um elemento ou atributo e excluindo subárvores. O seguinte exemplo exclui todos os elementos `chefvendas` dos anos anteriores até o ano dado por uma variável ou parâmetro SQL com o nome `@year`:

```
update RelatóriosViagem
set Relatório.modify(
  declare namespace c = "urn:exemplo/cliente";
  delete /c:doc/c:cliente//c:chefevendas[@year <
sql:variable("@year")]')
```

Execução de expressões XQuery

Como mencionado anteriormente, os dados XML são armazenados em uma representação binária interna. Entretanto, para executar as expressões XQuery, o tipo de dados XML é internamente transformado em uma chamada tabela de nó. A tabela de nó interna basicamente usa uma linha para representar um nó. Cada nó recebe um identificador OrdPath como seu ID de nó (um identificador OrdPath é um esquema de numeração decimal Dewey modificado; veja as notas bibliográficas para obter referências para mais informações sobre OrdPath). Cada nó também contém as informações-chave a fim de apontar novamente para a linha SQL original à qual o nó pertence, além de informações sobre o nome e tipo (em uma forma de ficha), valores etc. Como OrdPath codifica as informações de ordem do documento e de hierarquia, a tabela de nós, então, é agrupada com base nas informações-chave e em OrdPath, de modo que uma expressão de caminho ou recomposição de uma subárvore possa ser obtida com uma simples varredura de tabela.

Todas as expressões XQuery e update são, então, traduzidas para uma árvore de operador algébrico contra sua tabela de nós interna; a árvore usa os operadores relacionais comuns e alguns operadores especificamente projetados para a algebrização XQuery. A árvore resultante é, então, grafada para a árvore de álgebra da expressão relacional de modo que, no final, o mecanismo de execução de consulta receba uma única árvore de execução que ela possa otimizar e executar. Para evitar transformações onerosas em tempo de execução, um usuário pode pré-materializar a tabela de nós usando o índice XML primário. Além disso, o SQL Server 2005 fornece três índices XML secundários para que a execução de consulta possa tirar mais proveito das estruturas de índice:

- O índice *path* fornece suporte para tipos simples de expressões de caminho.
- O índice *properties* fornece suporte para o cenário comum de comparações com valor de propriedade.
- O índice *value* é adequado se a consulta usar curingas em comparações.

Veja nas notas bibliográficas as referências para mais informações sobre indexação XML e processamento de consulta no SQL Server 2005.

O Service Broker do SQL Server

O Service Broker ajuda os desenvolvedores a criar aplicações distribuídas frouxamente conectadas oferecendo suporte para geração de mensagens confiáveis em fila no SQL Server. Muitas aplicações de banco de dados usam processamento assíncrono para melhorar a capacidade de redimensionamento e os tempos de resposta para sessões interativas. Um método comum para o processamento assíncrono é usar tabelas de trabalho. Em vez de realizar todo o trabalho para um processo comercial em uma única transação de banco de dados, uma aplicação faz uma mudança indicando que um trabalho significativo está presente e, depois, insere um registro do trabalho a ser realizado em uma tabela de trabalho. Conforme os recursos permitirem, a aplicação processa a tabela de trabalho e completa o processo comercial. O Service Broker é uma parte do servidor de banco de dados que apóia diretamente esse método para o desenvolvimento de aplicação. A linguagem Transact-SQL inclui instruções DDL e DML para o Service Broker.

As tecnologias de enfileiramento de mensagens anteriores se concentravam em mensagens individuais. Com o Service Broker, a unidade básica de comunicação é a *conversação* – um fluxo de mensagens full-duplex persistente e seguro. O SQL Server garante que as mensagens dentro de uma conversação sejam distribuídas para uma aplicação exatamente uma vez em ordem. Cada conversação é parte de um *grupo de conversação*. As conversações relacionadas podem estar associadas ao mesmo grupo de conversação. As conversações ocorrem entre dois serviços. Um *serviço* é um lado nomeado para uma conversação.

As conversações e mensagens são fortemente tipificadas. Cada mensagem possui um tipo específico. O SQL Server pode opcionalmente validar que as mensagens sejam XML bem formadas, estejam vazias ou correspondam a um esquema XML. Um *contrato* define os tipos de mensagem que são permitidas para uma conversação e qual participante na conversação pode enviar mensagens de que tipo. O SQL Server fornece um tipo de contrato e mensagem padrão para aplicações que precisam apenas de um fluxo seguro.

O SQL Server armazena mensagens em tabelas internas. Essas tabelas não são diretamente acessíveis; em vez disso, o SQL Server expõe *filas* como views dessas tabelas internas. As aplicações recebem mensagens de uma fila. Uma operação receive retorna uma ou mais mensagens do mesmo grupo de conversação. Controlando o acesso à tabela básica, o SQL Server pode impor eficientemente a ordenação de mensagens, a correlação de mensagens relacionadas e o bloqueio. Como as filas são tabelas internas, elas não requerem qualquer tratamento especial para backup, restauração, recuperação de falhas ou espelhamento de banco de dados. Tanto as tabelas de aplicação quanto as mensagens associadas e em fila são copiadas em backup, restauradas e

recuperadas de falhas com o banco de dados. As conversações do Broker que existem em bancos de dados espelhados continuam onde pararam quando a recuperação de falha estiver completa – mesmo se a conversação estivesse entre dois serviços que residem em bancos de dados separados.

A granularidade de bloqueio para as operações do Service Broker é o grupo de conversação, em vez de uma conversação ou mensagens individuais específicas. Impondo bloqueio sobre o grupo de conversação, o Service Broker garante que apenas um leitor de fila de cada vez possa processar mensagens que pertencem a um determinado grupo de conversação. Isso elimina a necessidade de que a própria aplicação inclua lógica de prevenção de impasses – uma origem comum de erros em muitas aplicações de criação de mensagens. Outro efeito colateral saudável dessa semântica de bloqueio é que as aplicações podem escolher usar o grupo de conversação com uma chave para armazenar e recuperar estado de aplicação. Esses benefícios do modelo de programação são apenas dois exemplos das vantagens derivadas da decisão de formalizar a conversação como a primitiva de comunicação ou como a primitiva de mensagem atômica encontrada nos sistemas tradicionais de enfileiramento de mensagens.

O SQL Server pode ativar automaticamente procedimentos armazenados quando uma fila contém mensagens a serem processadas. Para dimensionar o número de procedimentos armazenados em execução para o tráfego de entrada, a lógica de ativação monitora a fila de modo a ver se existe trabalho útil para outro leitor de fila. O SQL Server considera a taxa em que os leitores existentes recebem mensagens e o número de grupos de conversação disponíveis para decidir quando iniciar outro leitor de fila. O procedimento armazenado a ser ativado, o contexto de segurança do procedimento armazenado e o número máximo de instâncias a serem iniciadas são configurados para uma fila individual. O SQL Server também expõe um evento que as aplicações externas podem usar para iniciar leitores de fila.

Como uma extensão lógica para a criação de mensagens assíncronas dentro da instância, o Service Broker também fornece criação segura de mensagens entre instâncias do SQL Server para permitir que desenvolvedores construam facilmente aplicações distribuídas. As conversações podem ocorrer dentro de uma única instância do SQL Server ou entre duas instâncias do SQL Server. As conversações locais e remotas usam o mesmo modelo de programação.

A segurança e o roteamento são configurados de maneira declarativa, sem exigir mudanças nos leitores de fila. O SQL Server usa *rotas* a fim de mapear um nome de serviço para o endereço de rede do outro participante na conversação. O SQL Server também pode realizar encaminhamento de mensagens e equilíbrio de carga simples para conversações. O SQL Server fornece distribuição segura, ordenada e exatamente uma vez, independente do número de instâncias pelas quais uma mensagem viaja. Uma conversação que abrange instâncias do SQL Server pode ser protegida tanto no nível de rede (não hierárquico) quanto no nível de conversação (ponta a ponta). Quando a segurança ponta a ponta é usada, o conteúdo da mensagem permanece criptografado até que a mensagem atinja o destino final, enquanto os cabeçalhos estão disponíveis para cada instância do SQL Server pela qual a mensagem viaja. As permissões-padrão do SQL Server se aplicam dentro de uma instância. A criptografia ocorre quando as mensagens deixam uma instância.

O SQL Server usa um protocolo binário para enviar mensagens entre instâncias. O protocolo fragmenta as mensagens grandes e permite fragmentos intercalados de múltiplas mensagens. A fragmentação permite que o SQL Server transmita rapidamente mensagens menores mesmo em casos em que uma mensagem grande está no processo de ser transmitida. O protocolo binário não usa transações distribuídas ou confirmação de duas fases. Em vez disso, o protocolo exige que um destinatário reconheça os fragmentos de mensagens. O SQL Server simplesmente repete a tentativa de enviar fragmentos de mensagem periodicamente até que o recebimento do fragmento seja reconhecido pelo destinatário. Os reconhecimentos são freqüentemente incluídos como parte dos cabeçalhos de uma mensagem de retorno, embora mensagens de retorno dedicadas sejam usadas se nenhuma mensagem de retorno estiver disponível.

Data warehouse e inteligência empresarial

Os componentes de data warehouse e inteligência empresarial do SQL Server contêm três subcomponentes:

- SQL Server Integration Services (SSIS), anteriormente chamado Data Transformation Services (DTS), que fornece o meio de integrar dados de várias origens, realizar transformações relacionadas à limpeza dos dados e à sua passagem para uma forma comum, e ao carregamento dos dados em um sistema de banco de dados.
- SQL Server Analysis Services (SSAS), que fornece capacidades de OLAP e mineração de dados.
- SQL Server Reporting Services (SSRS).

Os serviços de integração, de análise e de relatório são implementados em servidores separados e podem ser instalados independentemente um do outro na mesma máquina ou em máquinas diferentes. Eles podem se conectar a várias origens de dados, como um arquivo plano, planilhas ou diversos sistemas de banco de dados relacional, através de conectores nativos, drivers OLE-DB ou drivers ODBC.

Juntos, eles fornecem uma solução de ponta a ponta para extrair, transformar e carregar dados, modelando e acrescentando capacidade analítica aos dados e, finalmente, construindo e distribuindo relatórios sobre os dados. Os diferentes componentes do servidor de análise podem integrar e solicitar a capacidade uns dos outros. Aqui estão alguns cenários comuns que acionarão uma combinação de componentes:

- Construa um pacote SSIS que limpe dados, usando padrões gerados pela mineração de dados do SSAS.
- Use o SSIS para carregar dados em um cubo SSAS, processá-los e executar relatórios no cubo SSAS.
- Construa um relatório SSRS para publicar as descobertas de um modelo de mineração ou os dados contidos no componente OLAP do SSAS.

As seções a seguir fornecem um resumo das capacidades e da arquitetura de cada um desses componentes de servidor.

SQL Server Integration Services

O SQL Server 2005 Integration Services (SSIS) é uma solução de transformação e integração de dados empresariais que você pode usar para extrair, transformar, agregar e consolidar dados de origens diferentes, além de movê-los para um único ou vários destinos. Você pode usar o SSIS para realizar as seguintes tarefas:

- Mesclar dados de origens de dados heterogêneas.
- Atualizar dados em data warehouses e data marts.
- Limpar dados antes de carregá-los para os destinos.
- Carregar dados em massa em bancos de dados de processamento de transação on-line (OLTP) e processamento analítico on-line (OLAP).
- Enviar modificações.
- Automatizar funções administrativas.

O SSIS fornece um conjunto completo de serviços, ferramentas gráficas, objetos programáveis e APIs para essas tarefas. Elas fornecem a capacidade de construir soluções de transformação de dados grandes, robustas e complexas sem qualquer programação personalizada. Entretanto, uma API e objetos programáveis estão disponíveis, quando necessário, para criar elementos personalizados ou integrar capacidades de transformação de dados às aplicações personalizadas.

O mecanismo de fluxo de dados do SSIS fornece buffers na memória que movem dados da origem para o destino e chama os adaptadores de origem que extraem dados dos arquivos e bancos de dados relacionais. O mecanismo também oferece as transformações que modificam dados e os adaptadores de destino que carregam dados nos depósitos de dados. A eliminação duplicada baseada em correspondência aproximada é um exemplo de uma transformação fornecida pelo SSIS. Os usuários podem programar suas próprias transformações se necessário. A Figura 29.7 mos-

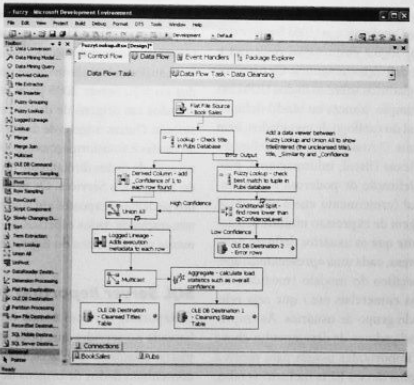


Figura 29.7 Carregamento de dados usando pesquisa aproximada.

tra um exemplo de como várias transformações podem ser combinadas para limpar e carregar informações de vendas de livros; os títulos de livro dos dados de vendas são comparados com um banco de dados de publicações, e, no caso de não haver qualquer coincidência, uma pesquisa aproximada é realizada para manipular títulos com pequenos erros (como erros de ortografia). Informações sobre o sigilo e a procedência dos dados são armazenadas com os dados impos.

SQL Server Analysis Services

O componente Analysis Services distribui funcionalidade de processamento analítico on-line (OLAP) e mineração de dados para aplicações de inteligência empresarial. O Analysis Services aceita uma arquitetura de cliente magro. O mecanismo de cálculo está no servidor e, portanto, as consultas são resolvidas no servidor, evitando a necessidade de transferir grandes quantidades de dados entre o cliente e o servidor.

QL Server Analysis Services: OLAP

O Analysis Services 2005 introduz um modelo dimensional nificado (UDM) que fecha a separação entre o relatório recional tradicional e a análise provisória OLAP. A função e um UDM é fornecer uma ponte entre o usuário e as oriens de dados. Um UDM é construído sobre uma ou mais rigens de dados físicas e, depois, o usuário emite consultas ara o UDM usando uma entre várias ferramentas de clien, como o Microsoft Excel.

Mais do que simplesmente uma camada de modelagem e dimensão dos esquemas DataSource, o UDM fornece um co ambiente para definir lógica empresarial poderosa e xaustiva, além de regras e definição semântica. Os usuários odem pesquisar e gerar relatórios sobre os dados UDM em u idioma nativo (por exemplo, francês ou hindi) definindo tradução de idioma local do catálogo de metadados, bem mo os dados dimensionais. O Analysis Services define diensões de tempo complexas (fiscal, informacional, protiva etc.) e permite a definição de poderosa lógica emesarial multidimensional (crescimento ano a ano, anual hoje) usando a linguagem de expressão multidimensiol (MDX). O UDM permite que os usuários definam perstivas orientadas à empresa, cada uma apresentando apes um subconjunto específico do modelo (medidas, dinsões, atributos, regras comerciais etc.) que seja relente para um determinado grupo de usuários. As empregeralmente definem indicadores de desempenho-chave PIs), que são métricas importantes usadas para medir a de da empresa. Exemplos desses KPIs incluem vendas, da por funcionário e taxa de fidelidade de cliente. O UDM permite que esses KPIs sejam definidos, possibilitando um agrupamento e uma apresentação de dados muito mais compreensível.

SQL Server Analysis Services: mineração de dados

O SQL Server 2005 fornece uma variedade de técnicas de mineração, com uma rica interface gráfica para ver resultados de mineração. Os algoritmos de mineração aceitos incluem:

- Regras de associação (úteis para aplicações de vendas cruzadas)
- Técnicas de classificação e previsão, como árvores de decisão, árvores de regressão, redes neurais e Bayes
- Projeção de série de tempo
- Técnicas de agrupamento, como maximização de expectativa e k-means (associadas a técnicas para agrupamento de seqüência).

O SQL Server também aceita as extensões da *Data-Mining Extensions* (DMX) para SQL. A DMX é a linguagem usada para interagir com modelos de mineração de dados, exatamente como a SQL é usada para interagir com tabelas e views. Como a DMX, os modelos podem ser criados e treinados e, em seguida, armazenados em um banco de dados do Analysis Services. O modelo, então, pode ser pesquisado para olhar padrões ou, usando uma sintaxe **prediction join** especial, aplicado em novos dados para realizar previsões. A linguagem DMX aceita funções e construções para determinar facilmente uma classe prevista juntamente com seu sigilo, prever uma lista dos itens associados como em um mecanismo de recomendação, ou mesmo retornar informações e fatos sustentadores sobre uma decisão. A mineração de dados no SQL Server 2005 pode ser usada com dados armazenados em origens de dados relacionais ou multidimensionais. Outras origens de dados também são aceitas através de tarefas e transformações especializadas, permitindo a mineração de dados diretamente no duto de dados operacional do Integration Services. Os resultados da mineração de dados podem ser expostos em controles gráficos, dimensões de mineração de dados especiais para cubos OLAP ou simplesmente nos relatórios do Reporting Services.

SQL Server Reporting Services

O Reporting Services é uma nova plataforma de relatório baseada no servidor que pode ser usada para criar e administrar relatórios tabulares, matriciais, gráficos e livres que contenham dados de origens de dados relacionais e multidimensionais. Os relatórios que você cria podem ser vistos

e administrados através de uma conexão baseada na Web. Os relatórios matriciais podem resumir dados para revisores de alto nível, ao mesmo tempo fornecendo detalhes de suporte em relatórios aprofundados. Os relatórios parametrizados podem ser usados para filtrar dados com base em valores que são fornecidos em tempo de execução. Os usuários podem escolher entre uma variedade de formatos de visualização para produzir relatórios dinamicamente nos formatos preferidos para manipulação de dados ou impressão. Uma API também está disponível para integrar ou estender capacidades de relatório em soluções personalizadas. A geração de relatórios baseada no servidor fornece uma maneira de centralizar o armazenamento e o gerenciamento de relatórios, definir políticas e acessos de segurança a relatórios e pastas, controlar como os relatórios são processados e distribuídos e padronizar a maneira como os relatórios são usados em sua empresa.

Notas bibliográficas

Informações detalhadas sobre um sistema certificado C2 com o SQL Server estão disponíveis em `www.microsoft.com/Downloads/Release.asp?ReleaseID=25503`.

A estrutura de otimização do SQL Server é baseada no protótipo de otimizador Cascades, proposto por Graefe [1995]. Simmen *et al.* [1996] discutem o esquema para reduzir colunas de agrupamento. Galindo-Legaria e Joshi [2001] apresentam a variedade de estratégias de execução que o SQL Server considera durante a otimização baseada em custo. Informações adicionais sobre aspectos de auto-ajuste do SQL Server são discutidas por Chaudhuri *et al.* [1999]. Chaudhuri e Shim [1994] e Yan e Larson [1995] discutem a agregação parcial.

Chatziantoniou e Ross [1997] e Galindo-Legaria e Joshi [2001] propuseram a alternativa usada pelo SQL Server para consultas SQL exigindo um self-join. Sob esse esquema, o otimizador detecta o padrão e considera a execução por segmento. Pellenkoft *et al.* [1997] discutem o esquema de otimização para gerar o espaço de pesquisa completo usando um conjunto de transformações que sejam completas, locais e não redundantes. Graefe *et al.* [1998] oferecem discussão concernente às operações de hash que aceitam agregação e junção básicas, com diversas otimizações, extensões e ajuste dinâmico para inclinação de dados. Graefe *et al.* [1998] apresentam a idéia de juntar índices com o único fim de montar uma linha com o conjunto de colunas necessárias em uma consulta. Ele argumenta que isso às vezes é mais rápido do que varrer uma tabela base.

Blakeley [1996] e Blakeley e Pizzo [2001] oferecem discussões a respeito da comunicação com o mecanismo de armazenamento através de OLE-DB. Blakeley *et al.* [2005] detalham a implementação das capacidades de consulta distribuídas e heterogêneas do SQL Server. Acheson *et al.* [2004] fornecem detalhes sobre a integração do CLR .NET dentro do processo do SQL Server.

O padrão SQL:2003 é definido em SQL/XML [2004]. Rys [2001] fornece mais detalhes sobre a funcionalidade XML do SQL Server 2000. Rys [2004] oferece um resumo das extensões para a agregação **for xml**. Para obter informações sobre as capacidades de XML que podem ser usadas no lado do cliente ou dentro do CLR, consulte a coleção de documentos no MSDN: XML Developer Center. O modelo de dados XQuery 1.0/XPath 2.0 é definido em Walsh *et al.* [2004]. Rys [2003] fornece uma sinopse das técnicas de implementação para o XQuery no contexto dos bancos de dados relacionais. O esquema de numeração OrdPath é descrito em O'Neil *et al.* [2004]; Pal *et al.* [2004] e Baras *et al.* [2005] oferecem mais informações sobre a indexação XML e a algebralização e otimização XQuery no SQL Server 2005.

Bibliografia

[Abiteboul *et al.* 1995] S. Abiteboul, R. Hull e V. Vianu, *Foundations of Databases*, Addison Wesley (1995).

[Abiteboul *et al.* 2003] S. Abiteboul, R. Agrawal, P. A. Bernstein, M. J. Carey *et al.* "The Lowell Database Research Self Assessment". CoRR cs.DB/0310006 (2003).

[Acharya *et al.* 1995] S. Acharya, R. Alonso, M. Franklin e S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 19-210. Também aparece em Imielinski e Korth [1996], Capítulo 12.

[Acheson *et al.* 2004] A. Acheson, M. Bendixen, J. A. Blakeley, I. P. Carlin, E. Ersan, J. Fang, X. Jiang, C. Kleinerman, B. Rathakrishnan, G. Schaller, B. Sezgin, R. Venkatesh e H. Zhang, "Hosting the .NET Runtime in Microsoft SQL Server", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004).

[Adali *et al.* 1996] S. Adali, K. S. Candan, Y. Papakonstantinou e V. S. Subrahmanian, "Query Caching and Optimization in Distributed Mediator Systems", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 137-148.

[Agarwal *et al.* 1996] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan e S. Sarawagi, "On the Computation of Multidimensional Attributes", *Proc. of the International Conf. on Very Large Databases*, Bombay, India (1996), páginas 506-521.

[Agrawal e Srikant 1994] R. Agrawal e R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases", *Proc. of the International Conf. on Very Large Databases* (1994), páginas 487-499.

[Agrawal *et al.* 1992] R. Agrawal, S. P. Ghosh, T. Imielinski, B. R. Iyer e A. N. Swami, "An Interval Classifier for Database Mining Applications", *Proc. of the International Conf. on Very Large Databases* (1992), páginas 560-573.

[Agrawal *et al.* 1993a] R. Agrawal, T. Imielinski e A. Swami, "Mining Association Rules between Sets of Items in Large Databases", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[Agrawal *et al.* 1993b] R. Agrawal, T. Imielinski e A. N. Swami, "Database Mining: A Performance Perspective", *IEEE Transactions on Knowledge and Data Engineering*, Volume 5, Número 6 (1993), páginas 914-925.

[Agrawal *et al.* 2000] S. Agrawal, S. Chaudhuri e V. R. Narasayya, "Automated Selection of Materialized Views and Indexes in SQL Databases", *Proc. of the International Conf. on Very Large Databases* (2000), páginas 496-505.

[Agrawal *et al.* 2002] S. Agrawal, S. Chaudhuri e G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases", *Proc. of the International Conf. on Data Engineering* (2002).

[Agrawal *et al.* 2004] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya e M. Syamala, "Database Tuning Advisor for Microsoft SQL Server 2005", *Proc. of the International Conf. on Very Large Databases* (2004).

[Aho *et al.* 1979a] A. V. Aho, C. Beeri e J. D. Ullman, "The Theory of Joins in Relational Databases", *ACM Transactions on Database Systems*, Volume 4, Número 3 (1979), páginas 297-314.

[Aho *et al.* 1979b] V. Aho, Y. Sagiv e J. D. Ullman, "Equivalences among Relational Expressions", *SIAM Journal of Computing*, Volume 8, Número 2 (1979), páginas 218-246.

[Aho *et al.* 1986] A. V Aho, R. Sethi e J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley (1986).

[Ailamaki *et al.* 2001] A. Ailamaki, D. J. DeWitt, M. D. Hill e M. Skounakis, "Weaving Relations for Cache Performance", *Proc. of the International Conf. on Very Large Databases* (2001), páginas 169-180.

[Alonso e Korth 1993] R. Alonso e H. F. Korth, "Database System Issues in Nomadic Computing", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), páginas 388-392.

[Amer-Yahia *et al.* 2004] S. Amer-Yahia, C. Botev e J. Shanmugasundaram, "TeXQuery: A Full-Text Search Extension to XQuery", *Proc. of the International World Wide Web Conf.* (2004).

[Anderson *et al.* 1992] D. P. Anderson, Y. Osawa e R. Govindan, "A File System for Continuous Media", *ACM Transactions on Database Systems*, Volume 10, Número 4 (1992), páginas 311-337.

[Anderson *et al.* 1998] T. Anderson, Y. Breitbart, H. F. Korth e A. Wool, "Replication, Consistency and Practicality: Are These Mutually Exclusive?", *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seattle, WA (1998).

[ANSI 1986] *American National Standard for Information Systems: Database Language SQL*. American National Standards Institute. FDT, ANSI X3.135-1986(1986).

[ANSI 1989] *Database Language SQL with Integrity Enhancement, ANSI X3, 135-1989*. American National Standards Institute, New York. Também disponível como ISO/IEC Document 9075:1989 (1989).

[ANSI 1992] *Database Language SQL, ANSI X3.135-1992*. American National Standards Institute, New York. Também disponível como ISO/IEC Document 9075:1992(1992).

[Antoshenkov 1995] G. Antoshenkov, "Byte aligned Bitmap Compression (poster abstract)", *IEEE Data Compression Conf.* (1995).

[Appelt e Israel 1999] D. E. Appelt e D. J. Israel, "Introduction to Information Extraction Technology", *IjCAI* (1999). Tutorial apresentado em IJCAI-99, disponível em http://www.ai.sri.com/appelt/ie-tutorial/ IJCAI99.pdf.

[Apt e Pugin 1987] K. R. Apt e J. M. Pugin, "Maintenance of Stratified Database Viewed as a Belief Revision System", *Proc. of the ACM Symposium on Principles of Database Systems* (1987), páginas 136-145.

[Armstrong 1974] W. W. Armstrong, "Dependency Structures of Data Base Relationships", *Proc. of the 1974 IFIP Congress* (1974), páginas 580-583.

[Astrahan *et al.* 1976] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray P. P. Griffiths, W. F. King, R. A. Lorie, P. R. Mcjones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade e V. Watson, "System R, A Relational Approach to Data Base Management", *ACM Transactions on Database Systems*, Volume 1, Número 2 (1976), páginas 97-137.

[Atreya *et al.* 2002] M. Atreya, B. Hammond, S. Paine, P. Starrett e S. Wu, *Digital Signatures*, RSA Press (2002).

[Atzeni e Antonellis 1993] P. Atzeni e V. D. Antonellis, *Relational Database Theory*, Benjamin Cummings, Menlo Park (1993).

[Baeza-Yates e Ribeiro-Neto 1999] R. Baeza Yates e B. Ribeiro-Neto, *Modern Information Retrieval*, Addison Wesley (1999).

[Bancilhon e Buneman 1990] F. Bancilhon e P. Buneman, *Advances in Database Programming languages*, ACM Press, New York (1990).

[Bancilhon *et al.* 1989] F. Bancilhon, S. Cluet e C. Delobel, "A Query Language for the O_2 Object-Oriented Database", *Proc. of the Second Workshop on Database Programming Languages* (1989).

[Banerjee *et al.* 2000] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad e R. Murthy, "Oracle 8i - The XML Enabled Data Management System"," *Proc. of the International Conf. on Data Engineering* (2000), páginas 561-568.

[Baras *et al.* 2005] A. Baras, D. Churin, I. Cseri, T. Grabs, E. Kogan, S. Pal, M. Rys e O. Seeliger, "Implementing XQuery in a Relational Database System", a ser publicado (2005).

[Barbara e Imielinski 1994] D. Barbara e T. Imielinski, "Sleepers and Workaholics: Caching Strategies in Mobile Environments", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1994), páginas 1-12.

[Baru *et al.* 1995] C. Bam *et al.*, "DB2 Parallel Edition", *IBM Systems Journal*, Volume 34, Número 2 (1995), páginas 292-322.

[Bassiouni 1988] M. Bassiouni, "Single-site and Distributed Optimistic Protocols for Concurrency Control", *IEEE Transactions on Software Engineering*, Volume SE-14, Número 8 (1988), páginas 1071-1080.

[Batini *et al.* 1992] C. Batini, S. Ceri e S. Navathe, *Database Design: An Entity-Relationship Approach*, Benjamin Cummings, Redwood City (1992).

[Bayer 1972] R. Bayer, "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", *Acta Informatica*, Volume 1, Número 4 (1972), páginas 290-306.

[Bayer e McCreight 1972] R. Bayer e E. M. McCreight, "Organization and Maintenance of Large Ordered Indices", *Acta Informatica*, Volume 1, Número 3 (1972), páginas 173-189.

[Bayer e Schkolnick 1977] R. Bayer e M. Schkolnick, "Concurrency of Operating on B-trees", *Acta Informatica*, Volume 9, Número 1 (1977), páginas 1-21.

[Bayer e Unterauer 1977] R. Bayer e K. Unterauer, "Prefix B-trees", *ACM Transactions on Database Systems*, Volume 2, Número 1 (1977), páginas 11-26.

[Bayer *et al.* 1978] R. Bayer, R. M. Graham e G. Seegmuller, editores, *Operating Systems: An Advanced Course*, Springer Verlag (1978).

[Bayer *et al.* 1980] R. Bayer, H. Heller e A. Reiser, "Parallelism and Recovery in Database Systems", *ACM Transactions on Database Systems*, Volume 5, Número 2 (1980), páginas 139-156.

[Beckmann *et al.* 1990] N. Beckmann, H. P. Kriegel, R. Schneider e B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), páginas 322-331.

[Beeri *et al.* 1977] C. Beeri, R. Fagin e J. H. Howard, "A Complete Axiomatization for Functional and Multivalued Dependencies", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1977), páginas 47-61.

[Bello *et al.* 1998] R. G. Bello, K. Dias, A. Downing, J. F. Jr., W. D. Norcott, H. Sun, A. Witkowski e M. Ziauddin, "Materialized Views in Oracle", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 659-664.

[Bentley 1975] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", *Communications of the ACM*, Volume 18, Número 9 (1975), páginas 509-517.

[Berenson *et al.* 1995] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil e P. O'Neil, "A Critique of ANSI SQL Isolation Levels", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 1-10.

[Bernstein e Goodman 1980] P. A. Bernstein e N. Goodman, "Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems", *Proc. of the International Conf. on Very Large Databases* (1980), páginas 285-300.

[Bernstein e Goodman 1981] P. A. Bernstein e N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Survey*, Volume 13, Número 2 (1981), páginas 185-221.

[Bernstein e Goodman 1982] P. A. Bernstein e N. Goodman, "A Sophisticated Introduction to Distributed Database Concurrency Control", *Proc. of the International Conf. on Very Large Databases* (1982), páginas 62-76.

[Bernstein e Newcomer 1997] P. A. Bernstein e E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann (1997).

[Bernstein *et al.* 1983] P. A. Bernstein, N. Goodman e M. Y. Lai, "Analyzing Concurrency Control when User and System Operations Differ", *IEEE Transactions on Software Engineering*, Volume SE-9, Número 3 (1983), páginas 233-239.

[Bernstein *et al.* 1987] A. Bernstein, V. Hadzilacos e N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley (1987).

[Bernstein *et al.* 1998] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker e J. Ullman, "The Asilomar Report on Database Research", *ACM SIGMOD Record*, Volume 27, Número 4 (1998).

[Berson *et al.* 1995] S. Berson, L. Golubchik e R. R. Muntz, "Fault Tolerant Design of Multimedia Servers", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 364-375.

[Bhalotia *et al.* 2002] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti e S. Sudarshan, "Keyword Searching and Browsing in Databases using BANKS", *Proc. of the International Conf. on Data Engineering* (2002).

[Bharat e Henzinger 1998] K. Bharat e M. R. Henzinger, "Improved Algorithms for Topic Distillation in a Hyperlinked Environment", *Proc. of the ACM SIGIR Conf. on Research and Development in Information Retrieval* (1998), páginas 104-111.

[Bhattacharjee *et al.* 2003] B. Bhattacharjee, S. Padmanabhan, T. Malkemus, T. Lai, L. Cranston e M. Huras, "Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2", *Proc. of the International Conf. on Very Large Databases* (2003), páginas 963-974.

[Biliris e Orenstein 1994] A. Biliris e J. Orenstein, "Object Storage Management Architectures", Em *Dogac et al. [1994]*, páginas 185-200 (1994).

[Biskup *et al.* 1979] J. Biskup, U. Dayal e P. A. Bernstein, "Synthesizing Independent Database Schemas", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979), páginas 143-152.

[Bitton e Gray 1988] D. Bitton e J. N. Gray, "Disk Shadowing", *Proc. of the International Conf. on Very Large Databases* (1988), páginas 331-338.

[Bitton *et al.* 1983] D. Bitton, D. J. DeWitt e C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach", *Proc. of the International Conf. on Very Large Databases* (1983).

[Bjork 1973] L. A. Bjork, "Recovery Scenario for a DB/DC System", *Proc. of the ACM Annual Conf.* (1973), páginas 142-146.

[Blakeley 1996] J. A. Blakeley, "Data Access for the Masses through OLE DB", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 161-172.

[Blakeley e Pizzo 2001] J. A. Blakeley e M. Pizzo, "Enabling Component Databases with OLE DB", K. R. Dittrich e A. Geppert, editores, *Component Database Systems*, Morgan Kaufmann Publishers (2001), páginas 139-173.

[Blakeley *et al.* 1986] J. A. Blakeley, P. Larson e F. W. Tompa, "Efficiently Updating Materialized Views", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1986), páginas 61-71.

[Blakeley *et al.* 1989] J. Blakeley, N. Coburn e P. Larson, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates", *ACM Transac-*

tions on Database Systems, Volume 14, Número 3 (1989), páginas 369-400.

[**Blakeley *et al.* 2005**] J. A. Blakeley, C. Cunningham, N. Ellis, B. Rathakrishnan e M.-C. Wu, "Distributed/Heterogeneous Query Processing in Microsoft SQL Server", *Proc. of the International Conf. on Data Engineering* (2005).

[**Blasgen e Eswaran 1976**] M. W. Blasgen e K. P. Eswaran, "On the Evaluation of Queries in a Relational Database System", *IBM Systems Journal*, Volume 16, (1976), páginas 363-377.

[**Boral *et al.* 1990**] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith e P. Valduriez, "Prototyping Bubba, a Highly Parallel Database System", *IEEE Transactions on Knowledge and Data Engineering*, Volume 2, Número 1 (1990).

[**Boyce *et al.* 1975**] R. Boyce, D. D. Chamberlin, W. F. King e M. Hammer, "Specifying Queries as Relational Expressions", *Communications of the ACM*, Volume 18, Número 11 (1975), páginas 621-628.

[**Breese *et al.* 1998**] J. Breese, D. Heckerman e C. Kadie, "Empirical Analysis of Predictive Algorithms for Collaborative Filtering", *Procs. Conf. on Uncertainty in Artificial Intelligence*, Morgan Kaufmann (1998).

[**Breitbart *et al.* 1999a**] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri e A. Silberschatz, "Update Propagation Protocols For Replicated Databases", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1999), páginas 97-108.

[**Breitbart *et al.* 1999b**] Y. Breitbart, H. Korth, A. Silberschatz e S. Sudarshan, "Distributed Databases", *In Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons (1999).

[**Brin e Page 1998**] S. Brin e L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine", *Proc. of the International World Wide Web Conf.* (1998).

[**Brinkhoff *et al.* 1993**] T. Brinkhoff, H.-P. Kriegel e B. Seeger, "Efficient Processing of Spatial Joins Using R-trees", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), páginas 237-246.

[**Bruno *et al.* 2002**] N. Bruno, S. Chaudhuri e L. Gravano, "Top-k Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation", *ACM Transactions on Database Systems*, Volume 27, Número 2 (2002), páginas 153-187.

[**Buckley e Silberschatz 1983**] G. Buckley e A. Silberschatz, "Obtaining Progressive Protocols for a Simple Multiversion Database Model", *Proc. of the International Conf. on Very Large Databases* (1983), páginas 74-81.

[**Buckley e Silberschatz 1984**] G. Buckley e A. Silberschatz, "Concurrency Control in Graph Protocols by Using Edge Locks", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1984), páginas 45-50.

[**Buckley e Silberschatz 1985**] G. Buckley e A. Silberschatz, "Beyond Two-Phase Locking", *Journal of the ACM*, Volume 32, Número 2 (1985), páginas 314-326.

[**Bulmer 1979**] M. G. Bulmer, *Principles of Statistics*, Dover Publications (1979).

[**Burkhard 1976**] W. A. Burkhard, "Hashing and Trie Algorithms for Partial Match Retrieval", *ACM Transactions on Database Systems*, Volume 1, Número 2 (1976), páginas 175-187.

[**Burkhard 1979**] W. A. Burkhard, "Partial-match Hash Coding: Benefits of Redundancy", *ACM Transactions on Database Systems*, Volume 4, Número 2 (1979), páginas 228-239.

[**Cannan e Otten 1993**] S. Cannan e G. Otten, *SQL–The Standard Handbook*, McGraw Hill (1993).

[**Carey 1983**] M. J. Carey, "Granularity Hierarchies in Concurrency Control", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1983), páginas 156-165.

[**Carey e Kossmann 1998**] M. J. Carey e D. Kossmann, "Reducing the Braking Distance of an SQL Query Engine", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 158-169.

[**Carey *et al.* 1991**] M. Carey, M. Franklin, M. Livny e E. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1991).

[**Carey *et al.* 1993**] M. J. Carey, D. DeWitt e J. Naughton, "The OO7 Benchmark", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[**Carey *et al.* 1999**] M. J. Carey, D. D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman e N. Mattos, "O-O, What Have They Done to DB2?", *Proc. of the International Conf. on Very Large Databases* (1999), páginas 542-553.

[**Cattell 2000**] R. Cattell, editor, *The Object Database Standard: ODMG 3.0*, Morgan Kaufmann (2000).

[**Cattell e Skeen 1992**] R. Cattell e J. Skeen, "Object Operations Benchmark", *ACM Transactions on Database Systems*, Volume 17, Número 1 (1992).

[**Ceri e Owicki 1983**] S. Ceri e S. Owicki, "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", *Proc. of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks* (1983).

[**Ceri e Pelagatti 1984**] S. Ceri e G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw Hill (1984).

[**Chakrabarti 1999**] S. Chakrabarti, "Recent Results in Automatic Web Resource Discovery", *ACM Computing Surveys*, Volume 31, Número 4 (1999).

[**Chakrabarti 2000**] S. Chakrabarti, "Data Mining for Hypertext: A Tutorial Survey", *SIGKDD Explorations*, Volume 1, Número 2 (2000), páginas 1-11.

[Chakrabarti 2002] S. Chakrabarti, *Mining the Web: Discovering Knowledge from HyperText Data*, Morgan Kaufmann (2002).

[Chakrabarti *et al.* 1998] S. Chakrabarti, S. Sarawagi e B. Dom, "Mining Surprising Patterns Using Temporal Description Length", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 606-617.

[Chakrabarti *et al.* 1999] S. Chakrabarti, M. van den Berg e B. Dom, "Focused Crawling: A New Approach to Topic Specific Web Resource Discovery", *Proc. of the International World Wide Web Conf.* (1999).

[Chakravarthy *et al.* 1990] U. S. Chakravarthy, J. Grant e J. Minker, "Logic-Based Approach to Semantic Query Optimization", *ACM Transactions on Database Systems*, Volume 15, Número 2 (1990), páginas 162-207.

[Chamberlin 1996] D. Chamberlin, *Using the New DB2: IBM's Object-Relational Database System*, Morgan Kaufmann (1996).

[Chamberlin 1998] D. D. Chamberlin, *A Complete Guide to DB2 Universal Database*, Morgan Kaufmann (1998).

[Chamberlin e Boyce 1974] D. D. Chamberlin e R. F. Boyce, "SEQUEL: A Structured English Query Language", *ACM SIGMOD Workshop on Data Description, Access, and Control* (1974).

[Chamberlin *et al.* 1976] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner e B. W. Wade, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", *IBM Journal of Research and Development*, Volume 20, Número 6 (1976), páginas 560-575.

[Chamberlin *et al.* 1981] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade e R. A. Yost, "A History and Evaluation of System R", *Communications of the ACM*, Volume 24, Número 10 (1981), páginas 632-646.

[Chamberlin *et al.* 2000] D. D. Chamberlin, J. Robie e D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources", *Proc. of the International Workshop on the Web and Databases (WebDB)* (2000), páginas 53-62.

[Chan e Ioannidis 1998] C.-Y. Chan e Y. E. Ioannidis, "Bitmap Index Design and Evaluation", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1998).

[Chan e Ioannidis 1999] C.-Y. Chan e Y. E. Ioannidis, "An Efficient Bitmap Encoding Scheme for Selection Queries", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1999).

[Chandra e Harel 1982] A. K. Chandra e D. Harel, "Structure and Complexity of Relational Queries", *Journal of Computer and System Sciences*, Volume 15, Número 10 (1982), páginas 99-128.

[Chandrasekaran *et al.* 2003] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss e M. A. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World", *Proc. First Biennial Conference on Innovative Data Systems Research* (2003).

[Chandy *et al.* 1975] K. M. Chandy, J. C. Browne, C. W. Dissley e W. R. Uhrig, "Analytic Models for Rollback and Recovery Strategies in Database Systems", *IEEE Transactions on Software Engineering*, Volume SE-1, Número 1 (1975), páginas 100-110.

[Chatziantoniou e Ross 1997] D. Chatziantoniou e K. A. Ross, "Groupwise Processing of Relational Queries", *Proc. of the International Conf. on Very Large Databases* (1997), páginas 476-485.

[Chaudhuri e Narasayya 1997] S. Chaudhuri e V. Narasayya, "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server", *Proc. of the International Conf. on Very Large Databases* (1997).

[Chaudhuri e Shim 1994] S. Chaudhuri e K. Shim, "Including Group-By in Query Optimization", *Proc. of the International Conf. on Very Large Databases* (1994).

[Chaudhuri *et al.* 1995] S. Chaudhuri, R. Krishnamurthy, S. Potamianos e K. Shim, "Optimizing Queries with Materialized Views", *Proc. of the International Conf. on Data Engineering*, Taipei, Taiwan (1995).

[Chaudhuri *et al.* 1999] S. Chaudhuri, E. Christensen, G. Graefe, V. Narasayya e M. Zwilling, "Self Tuning Technology in Microsoft SQL Server", *IEEE Data Engineering Bulletin*, Volume 22, Número 2 (1999).

[Chaudhuri *et al.* 2003] S. Chaudhuri, K. Ganjam, V. Ganti e R. Motwani, "Robust and Efficient Fuzzy Match for Online Data Cleaning", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003).

[Chen 1976] P. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems*, Volume 1, Número 1 (1976), páginas 9-36.

[Chen e Patterson 1990] P. Chen e D. Patterson, "Maximizing Performance in a Striped Disk Array", *Proc. of the Seventeenth Annual International Symposium on Computer Architecture* (1990).

[Chen *et al.* 1994] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz e D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Survey*, Volume 26, Número 2 (1994).

[Chomicki 1995] J. Chomicki, "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding", *ACM Transactions on Database Systems*, Volume 20, Número 2 (1995), páginas 149-186.

[Chou e Dewitt 1985] H. T. Chou e D. J. Dewitt, "An Evaluation of Buffer Management Strategies for Relational

Database Systems", *Proc. of the International Conf. on Very Large Databases* (1985), páginas 127-141.

[**Cochrane et al. 1996**] R. Cochrane, H. Pirahesh e N. M. Mattos, "Integrating Triggers and Declarative Constraints in SQL Database Systems", *Proc. of the International Conf. on Very Large Databases* (1996).

[**Codd 1970**] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Volume 13, Número 6 (1970), páginas 377-387.

[**Codd 1972**] E. F. Codd. "Further Normalization of the Data Base Relational Model", Em *Rustin [1972]*, páginas 33-64 (1972).

[**Codd 1979**] E. F. Codd, "Extending the Database Relational Model to Capture More Meaning", *ACM Transactions on Database Systems*, Volume 4, Número 4 (1979), páginas 397-434.

[**Codd 1982**] E. F. Codd, "The 1981 ACM Turing Award Lecture: Relational Database: A Practical Foundation for Productivity", *Communications of the ACM*, Volume 25, Número 2 (1982), páginas 109-117.

[**Codd 1990**] E. F. Codd, *The Relational Model for Database Management: Version 2*. Addison Wesley (1990).

[**Comer 1979**] D. Comer, "The Ubiquitous B-tree", *ACM Computing Survey*, Volume 11, Número 2 (1979), páginas 121-137.

[**Comer e Droms 2003**] D. E. Comer e R. E. Droms, *Computer Networks and Internets*, 4ª edição, Prentice Hall (2003).

[**Cook 1996**] M. A. Cook, *Building Enterprise Information Architecture: Reengineering Information Systems*, Prentice Hall (1996).

[**Cook et al. 1999**] J. Cook, R. Harbus e T. Shirai, *The DB2 Universal Database V6.1 Certification Guide: For UNIX, Windows, and OS/2*, Prentice Hall (1999).

[**Cormen et al. 1990**] T. Cormen, C. Leiserson e R. Rivest, *Introduction to Algorithms*, MIT Press (1990).

[**Cruanes et al. 2004**] T. Cruanes, B. Dageville e B. Ghosh, "Parallel SQL Execution in Oracle 10g", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), páginas 850-854.

[**Dageville e Zait 2002**] B. Dageville e M. Zait, "SQL Memory Management in Oracle 9i", *Proc. of the International Conf. on Very Large Databases* (2002), páginas 962-973.

[**Dageville et al. 2004**] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait e M. Ziauddin, "Automatic SQL Tuning in Oracle 10g", *Proc. of the International Conf. on Very Large Databases* (2004), páginas 1098-1109.

[**Dalvi et al. 2001**] N. N. Dalvi, S. K. Sanghai, P. Roy e S. Sudarshan, "Pipelining in Multi-Query Optimization", *Proc. of the ACM Symposium on Principles of Database Systems* (2001).

[**Daniels et al. 1982**] D. Daniels, P. G. Selinger, L. M. Haas, B. G. Lindsay, C. Mohan, A. Walker e P. F. Wilms, "An Introduction to Distributed Query Compilation in R*", *In Schneider [1982]* (1982).

[**Dar et al. 1996**] S. Dar, H. V. Jagadish, A. Levy e D. Srivastava, "Answering Queries with Aggregation Using Views", *Proc. of the International Conf. on Very Large Databases* (1996).

[**Dashti et al. 2003**] A. Dashti, S. H. Kim, C. Shahabi e R. Zimmermann, *Streaming Media Server Design*, Prentice Hall (2003).

[**Date 1986**] C. J. Date, *Relational Databases: selected Writings*, Addison Wesley (1986).

[**Date 1989**] C. Date, *A Guide to DB2*, Addison Wesley (1989).

[**Date 1990**] C. J. Date, *Relational Database Writings, 1985-1989*, Addison Wesley (1990).

[**Date 1993a**] C. J. Date, "How SQL Missed the Boat", *Database Programming and Design*, Volume 6, Número 9 (1993).

[**Date 1993b**] C. J. Date, "The Outer Join", *IOCOD-2*, John Wiley and Sons (1993), páginas 76-106.

[**Date 2003**] C. J. Date, *An Introduction to Database Systems*, 8ª edição, Addison Wesley (2003).

[**Date e Darwen 1997**] C. J. Date e G. Darwen, *A Guide to the SQL Standard*, 4ª edição, Addison Wesley (1997).

[**Davies 1973**] C. T. Davies, "Recovery Semantics for a DB/DC System", *ACM Annual Conference* (1973), páginas 136-141.

[**Davis et al. 1983**] C. Davis, S. Jajodia, P. A. Ng e R. Yeh, editores, *Entity-Relationship Approach to Software Engineering*, North Holland (1983).

[**Davison e Graefe 1994**] D. L. Davison e G. Graefe, "Memory-Contention Responsive Hash Joins", *Proc. of the International Conf. on Very Large Databases* (1994).

[**Dayal 1987**] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates and Quantifiers", *Proc. of the International Conf. on Very Large Databases* (1987), páginas 197-208.

[**Dayal et al. 1982**] U. Dayal, N. Goodman e R. H. Katz, "An Extended Relational Algebra with Control over Duplicate Elimination", *Proc. of the ACM Symposium on Principles of Database Systems* (1982).

[**DB2 Online documentation**] Documento on-line do DB2. www.software.ibm.com/db2/pubs.

[**Deshpande e Larson 1992**] V. Deshpande e P. A. Larson, "The Design and Implementation of a Parallel Join Algorithm for Nested Relations on Shared-Memory Multiprocessors", *Proc. of the International Conf. on Data Engineering* (1992).

[**Deutsch et al. 1999**] A. Deutsch, M. Fernandez, D. Florescu, A. Levy e D. Suciu, "A Query Language for XML",

Proc. of the International World Wide Web Conf. (1999). (XML-QL also submitted to the World Wide Web Consortium http://www.w3.org/TR/1998/NOTE-xml-ql-19980819).

[DeWitt 1990] D. DeWitt, "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, Volume 2, Número 1 (1990).

[DeWitt e Gray 1992] D. DeWitt e J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Communications of the ACM*, Volume 35, Número 6 (1992), páginas 85-98.

[DeWitt *et al.* 1986] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar e M. Muralikrishna, "A High Performance Dataflow Database Machine", *Proc. of the International Conf. on Very Large Databases* (1986).

[DeWitt *et al.* 1992] D. DeWitt, J. Naughton, D. Schneider e S. Seshadri, "Practical Skew Handling in Parallel Joins", *Proc. of the International Conf. on Very Large Databases* (1992).

[Dias *et al.* 1989] D. Dias, B. Iyer, J. Robinson e P. Yu, "Integrated Concurrency-Coherency Controls for Multisystem Data Sharing", *Software Engineering*, Volume 15, Número 4 (1989), páginas 437-448.

[Dogac *et al.* 1994] A. Dogac, M. T. Ozsu, A. Biliris e T. Selis, *Advances in Object-Oriented Database Systems*, volume 130, Springer Verlag, Computer and Systems Sciences, NATO ASI Series F (1994).

[Donahoo e Speegle 2005] M. J. Donahoo e G. D. Speegle, *SQL: Practical Guide for Developers*, Morgan Kaufmann (2005).

[Douglas e Douglas 2003] K. Douglas e S. Douglas, *PostgreSQL*, Sam's Publishing (2003).

[Dubois e Thakkar 1992] M. Dubois e S. Thakkar, editores, *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers (1992).

[Duncan 1990] R. Duncan, "A Survey of Parallel Computer Architectures", *IEEE Computer*, Volume 23, Número 2 (1990), páginas 5-16.

[Eisenberg e Melton 1999] A. Eisenberg e J. Melton, "SQL:1999, formerly known as SQL3", *ACM SIGMOD Record*, Volume 28, Número 1 (1999).

[Eisenberg e Melton 2004a] A. Eisenberg e J. Melton, "Advancements in SQL/XML", *ACM SIGMOD Record*, Volume 33, Número 3 (2004), páginas 79-86.

[Eisenberg e Melton 2004b] A. Eisenberg e J. Melton, "An Early Look at XQuery API for Java (XQJ)", *ACM SIGMOD Record*, Volume 33, Número 2 (2004), páginas 105-111.

[Eisenberg *et al.* 2004] A. Eisenberg, J. Melton, K. G. Kulkarni, J.-E. Michels e F. Zemke, "SQL:2003 Has Been Published", *ACM SIGMOD Record*, Volume 33, Número 1 (2004), páginas 119-126.

[Ellis 1987] C. S. Ellis, "Concurrency in Linear Hashing", *ACM Transactions on Database Systems*, Volume 12, Número 2 (1987), páginas 195-217.

[Elmasri e Navathe 2003] R. Elmasri e S. B. Navathe, *Fundamentals of Database Systems*, 4ª edição, Addison Wesley (2003).

[Eppinger *et al.* 1991] J. L. Eppinger, L. B. Mummert e A. Z. Spector, *Camelot and Avalon: A Distributed Transaction Facility*, Morgan Kaufmann(1991).

[Epstein e Stonebraker 1980] R. Epstein e M. R. Stonebraker, "Analysis of Distributed Database Processing Strategies", *Proc. of the International Conf. on Very Large Databases* (1980), páginas 92-110.

[Epstein *et al.* 1978] R. Epstein, M. R. Stonebraker e E. Wong, "Distributed Query Processing in a Relational Database System", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1978), páginas 169-180.

[Escobar-Molano *et al.* 1993] M. Escobar-Molano, R. Hull e D. Jacobs, "Safety and Translation of Calculus Queries with Scalar Functions", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), páginas 253-264.

[Eswaran *et al.* 1976] K. P. Eswaran, J. N. Gray, R. A. Lorie e I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Volume 19, Número 11 (1976), páginas 624-633.

[Fagin 1977] R. Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases", *ACM Transactions on Database Systems*, Volume 2, Número 3 (1977), páginas 262-278.

[Fagin 1979] R. Fagin, "Normal Forms and Relational Database Operators", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979).

[Fagin 1981] R. Fagin, "A Normal Form for Relational Databases That Is Based on Domains and Keys", *ACM Transactions on Database Systems*, Volume 6, Número 3 (1981), páginas 387-415.

[Fagin *et al.* 1979] R. Fagin, J. Nievergelt, N. Pippenger e H. R. Strong, "Extendible Hashing – A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems*, Volume 4, Número 3 (1979), páginas 315-344.

[Faloutsos e Lin 1995] C. Faloutsos e K.-I. Lin, "Fast Map: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 163-174.

[Fayyad *et al.* 1995] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth e R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, MIT Press (1995).

[Fekete *et al.* 1990] A. Fekete, N. Lynch, M. Merritt e W. Weihl, "Commutativity-Based Locking for Nested Transactions", *Journal of Computer and System Science* (1990), páginas 65-156.

[Fekete et al. 2005] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil e D. Shasha, "Making Snapshot Isolation Serializable", *ACM Transactions on Database Systems*, Volume 30, Número 2 (2005).

[Finkel e Bentley 1974] R. A. Finkel e J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys", *Acta Informatica*, Volume 4, (1974), páginas 1-9.

[Finkelstein 1982] S. Finkelstein, "Common Expression Analysis in Database Applications", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1982), páginas 235-245.

[Fischer 2001] L. Fischer, editor, *Workflow Handbook 2001*, Future Strategies (2001).

[Fischer 2004] L. Fischer, editor, *The Workflow Handbook 2004*, Future Strategies Inc. (2004).

[Florescu e Kossmann 1999] D. Florescu e D. Kossmann, "Storing and Querying XML Data Using an RDBMS", *IEEE Data Engineering Bulletin (Special Issue on XML)* (1999).

[Florescu et al. 2000] D. Florescu, D. Kossmann e I. Manolescu, "Integrating Keyword Search into XML Query Processing", *Proc. of the International World Wide Web Conf.* (2000), páginas 119-135. Também aparece como uma edição especial de Computer Networks.

[Franklin et al. 1992] M. J. Franklin, M. J. Zwilling, C. K. Tan, M. J. Carey e D. J. DeWitt, "Crash Recovery in Client-Server EXODUS", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1992), páginas 165-174.

[Franklin et al. 1993] M. J. Franklin, M. Carey e M. Livny, "Local Disk Caching for Client-Server Database Systems", *Proc. of the International Conf. on Very Large Databases* (1993).

[Fredkin 1960] E. Fredkin, "Trie Memory", *Communications of the ACM*, Volume 4, Número 2 (1960), páginas 490-499.

[Freedman e DeWitt 1995] C. S. Freedman e D. J. DeWitt, "The SPIFFI Scalable Video-on-Demand Server", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 352-363.

[Funderburk et al. 2002a] J. E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita e C. Wei, "XTABLES: Bridging Relational Technology and XML", *IBM Systems Journal*, Volume 41, Número 4 (2002), páginas 616-641.

[Funderburk et al. 2002b] J. E. Funderburk, S. Malaika e B. Reinwald, "XML Programming with SQL/XML and XQuery", *IBM Systems Journal*, Volume 41, Número 4 (2002), páginas 642-665.

[Fushimi et al. 1986] S. Fushimi, M. Kitsuregawa e H. Tanaka, "An Overview of the Systems Software of a Parallel Relational Database Machine: GRACE", *Proc. of the International Conf. on Very Large Databases* (1986).

[Galindo-Legaria 1994] C. Galindo-Legaria, "Outerjoins as Disjunctions", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1994).

[Galindo-Legaria e Joshi 2001] C. A. Galindo-Legaria e M. Joshi, "Orthogonal Optimization of Subqueries and Aggregation", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2001).

[Galindo-Legaria e Rosenthal 1992] C. Galindo-Legaria e A. Rosenthal, "How to Extend a Conventional Optimizer to Handle One- and Two-Sided Outerjoin", *Proc. of the International Conf. on Data Engineering* (1992), páginas 402-409.

[Galindo-Legaria et al. 2004] C. Galindo-Legaria, S. Stefani e F. Waas, "Query Processing for SQL Updates", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), páginas 844-849.

[Ganguly 1998] S. Ganguly, "Design and Analysis of Parametric Query Optimization Algorithms", *Proc. of the International Conf. on Very Large Databases*, New York City, New York (1998).

[Ganguly et al. 1992] S. Ganguly, W. Hasan e R. Krishnamurthy, "Query Optimization for Parallel Execution", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1992).

[Ganguly et al. 1996] S. Ganguly, P. Gibbons, Y. Matias e A. Silberschatz, "A Sampling Algorithm for Estimating Join Size", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Ganski e Wong 1987] R. A. Ganski e H. K. T. Wong, "Optimization of Nested SQL Queries Revisited", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987).

[Garcia-Molina 1982] H. Garcia-Molina, "Elections in Distributed Computing Systems", *IEEE Transactions on Computers*, Volume C-31, Número 1 (1982), páginas 48-59.

[Garcia-Molina e Salem 1987] H. Garcia-Molina e K. Salem, "Sagas", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987), páginas 249-259.

[Garcia-Molina e Salem 1992] H. Garcia-Molina e K. Salem, "Main Memory Database Systems: An Overview", *IEEE Transactions on Knowledge and Data Engineering*, Volume 4, Número 6 (1992), páginas 509-516.

[Garcia-Molina et al. 2001] H. Garcia-Molina, J. D. Ullman e J. D. Widom, *Database Systems: The Complete Book*, Prentice Hall (2001).

[Gawlick 1998] D. Gawlick, "Messaging/Queuing in Oracle 8", *Proc. of the International Conf. on Data Engineering* (1998), páginas 66-68.

[Georgakopoulos et al. 1994] D. Georgakopoulos, M. Rusinkiewicz e A. Seth, "Using Tickets to Enforce the Serializability of Multidatabase Transactions", *IEEE*

Transactions on Knowledge and Data Engineering, Volume 6, Número 1 (1994), páginas 166-180.

[Gifford 1979] D. K. Gifford, "Weighted Voting for Replicated Data", *Proc. the ACM SIGOPS Symposium on Operating Systems Principles* (1979), páginas 150-162.

[Graefe 1990] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), páginas 102-111.

[Graefe 1993] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Survey*, Volume 25, Número 2 (1993), páginas 73-170.

[Graefe 1995] G. Graefe, "The Cascades Framework for Query Optimization", *Data Engineering Bulletin*, Volume 18, Número 3 (1995).

[Graefe e McKenna 1993] G. Graefe e W. McKenna, "The Volcano Optimizer Generator", *Proc. of the International Conf. on Data Engineering* (1993), páginas 209-218.

[Graefe et al. 1998] G. Graefe, R. Bunker e S. Cooper, "Hash Joins and Hash Teams in Microsoft SQL Server", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 86-97.

[Gray 1978] J. Gray. "Notes on Data Base Operating System". Em *Bayer et al. [1978]*, páginas 393-481 (1978).

[Gray 1981] J. Gray, "The Transaction Concept: Virtues and Limitations", *Proc. of the International Conf. on Very Large Databases* (1981).

[Gray 1991] J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*, 2ª edição, Morgan Kaufmann (1991).

[Gray e Edwards 1995] J. Gray e J. Edwards, "Scale Up with TP Monitors", *Byte*, Volume 20, Número 4 (1995), páginas 123-130.

[Gray e Graefe 1997] J. Gray e G. Graefe, "The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb", *SIGMOD Record*, Volume 26, Número 4 (1997).

[Gray e Putzolu 1987] J. Gray e G. R. Putzolu, "The 5 Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading Memory for CPU Time", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987), páginas 395-398.

[Gray e Reuter 1993] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

[Gray et al. 1975] J. Gray, R. A. Lorie e G. R. Putzolu, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", *Proc. of the International Conf. on Very Large Databases* (1975), páginas 428-451.

[Gray et al. 1976] J. Gray, R. A. Lorie, G. R. Putzolu e I. L. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, Nijssen (1976).

[Gray et al. 1981] J. Gray, P. R. Mcjones e M. Blasgen, "The Recovery Manager of the System R Database Manager", *ACM Computing Survey*, Volume 13, Número 2 (1981), páginas 223-242.

[Gray et al. 1990] J. Gray, B. Horst e M. Walker, "Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 148-161.

[Gray et al. 1995] J. Gray, A. Bosworth, A. Layman e H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Totals", Technical report, Microsoft Research (1995).

[Gray et al. 1996] J. Gray, P. Helland e P. O'Neil, "The Dangers of Replication and a Solution", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 173-182.

[Gray et al. 1997] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow e H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals", *Data Mining and Knowledge Discovery*, Volume 1, Número 1 (1997), páginas 29-53.

[Gregersen e Jensen 1999] H. Gregersen e C. S. Jensen, "Temporal Entity-Relationship Models-A Survey", *IEEE Transactions on Knowledge and Data Engineering*, Volume 11, Número 3 (1999), páginas 464-497.

[Griffin e Libkin 1995] T. Griffin e L. Libkin, "Incremental Maintenance of Views with Duplicates", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995).

[Grossman e Frieder 2004] D. A. Grossman e O. Frieder, *Information Retrieval: Algorithms and Heuristics*, 2ª edição, Springer Verlag (2004).

[Guo et al. 2003] L. Guo, F. Shao, C. Botev e J. Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003).

[Gupta 1997] H. Gupta, "Selection of Views to Materialize in a Data Warehouse", *Proc. of the International Conf. on Database Theory* (1997).

[Gupta e Mumick 1995] A. Gupta e I. S. Mumick, "Maintenance of Materialized Views: Problems, Techniques and Applications", *IEEE Data Engineering Bulletin*, Volume 18, Número 2 (1995).

[Guttman 1984] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1984), páginas 47-57.

[H. Lu e Tan 1991] M. S. H. Lu e K. Tan, "Optimization of Multi-Way Join Queries for Parallel Execution", *Proc. of the International Conf. on Very Large Databases* (1991).

[Haas et al. 1989] L. M. Haas, J. C. Freytag, G. M. Lohman e H. Pirahesh, "Extensible Query Processing in Star-

burst", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989), páginas 377-388.

[Haas *et al.* 1990] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey e E. J. Shekita, "Starburst Mid-Flight: As the Dust Clears", *IEEE Transactions on Knowledge and Data Engineering*, Volume 2, Número 1 (1990), páginas 143-160.

[Haerder e Reuter 1983] T. Haerder e A. Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Computing Survey*, Volume 15, Número 4 (1983), páginas 287-318.

[Haerder e Rothermel 1987] T. Haerder e K. Rothermel, "Concepts for Transaction Recovery in Nested Transactions", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987), páginas 239-248.

[Hall 1976] P. A. V. Hall, "Optimization of a Single Relational Expression in a Relational Database System", *IBM Journal of Research and Development*, Volume 20, Número 3 (1976), páginas 244-257.

[Halsall 1996] F. Halsall, *Data Communications, Computer Networks, and Open Systems*, 4ª edição, Addison Wesley (1996).

[Han e Kamber 2000] J. Han e M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann (2000).

[Harinarayan *et al.* 1996] V. Harinarayan, J. D. Ullman e A. Rajaraman, "Implementing Data Cubes Efficiently", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Haritsa *et al.* 1990] J. Haritsa, M. Carey e M. Livny, "On Being Optimistic about Real-Time Constraints", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990).

[Harizopoulos e Ailamaki 2004] S. Harizopoulos e A. Ailamaki, "STEPS towards Cache-resident Transaction Processing", *Proc. of the International Conf. on Very Large Databases* (2004).

[Hellerstein e Stonebraker 2005] J. M. Hellerstein e M. Stonebraker, editores, *Readings in Database Systems*, 4ª edição, Morgan Kaufmann (2005).

[Hellerstein *et al.* 1995] J. M. Hellerstein, J. E Naughton e A. Pfeffer, "Generalized Search Trees for Database Systems", *Proc. of the International Conf. on Very Large Databases* (1995).

[Hennessy *et al.* 2002] J. L. Hennessy, D. A. Patterson e D. Goldberg, *Computer Architecture: A Quantitative Approach*, 3ª edição, Morgan Kaufmann (2002).

[Hevner e Yao 1979] A. R. Hevner e S. B. Yao, "Query Processing in Distributed Database Systems", *IEEE Transactions on Software Engineering*, Volume SE-5, Número 3 (1979), páginas 177-187.

[Heywood *et al.* 2002] I. Heywood, S. Cornelius e S. Carver, *An Introduction to Geographical Information Systems, Second Edition*, Prentice Hall (2002).

[Hollingsworth 1994] D. Hollingsworth, *The Workflow Reference Model*, Workflow Management Coalition, TC00-1003 (1994).

[Hollingsworth 2004] D. Hollingsworth. "The Workflow Reference Model 10 Years On", Em Fischer [2004] (2004).

[Hong e Stonebraker 1991] W. Hong e M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS", *Proc. of the International Conf. on Parallel and Distributed Information Systems* (1991), páginas 218-225.

[Hong *et al.* 1993] D. Hong, T. Johnson e S. Chakravarthy, "Real-Time Transaction Scheduling: A Cost Conscious Approach", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[Howes *et al.* 1999] T. A. Howes, M. C. Smith e G. S. Good, *Understanding and Deploying LDAP Directory Services*, Macmillan Publishing, New York (1999).

[Hristidis e Papakonstantinou 2002] V. Hristidis e Y. Papakonstantinou, "DISCOVER: Keyword Search in Relational Databases", *Proc. of the International Conf. on Very Large Databases* (2002).

[Huang e Garcia-Molina 2001] Y. Huang e H. Garcia-Molina, "Exactly-once Semantics in a Replicated Messaging System", *Proc. of the International Conf. on Data Engineering* (2001), páginas 3-12.

[Huffman 1993] A. Huffman, "Transaction Processing with TUXEDO", *Proc. of the International Conf. on Parallel and Distributed Information Systems* (1993).

[Hulgeri e Sudarshan 2003] A. Hulgeri e S. Sudarshan, "AniPQO: Almost Non-Intrusive Parametric Query Optimization for Non-Linear Cost Functions", *Proc. of the International Conf. on Very Large Databases* (2003).

[IBM 1987] IBM, "Systems Application Architecture: Common Programming Interface, Database Reference", Technical report, IBM Corporation, IBM Form Number SC26-4348-0 (1987).

[IDEF1X1993] IDEF1X, "Integration Definition for Information Modeling (IDEF1X)", Technical Report Federal Information Processing Standards Publication 184, National Institute of Standards and Technology (NIST), Disponível em www.idef.com/Downloads/pdf/Idef1x.pdf (1993).

[Imielinski e Badrinath 1994] T. Imielinski e B. R. Badrinath, "Mobile Computing – Solutions and Challenges", *Communications of the ACM*, Volume 37, Número 10 (1994).

[Imielinski e Korth 1996] T. Imielinski e H. F. Korth, editores, *Mobile Computing*, Kluwer Academic Publishers (1996).

[Ioannidis e Christodoulakis 1993] Y. Ioannidis e S. Christodoulakis, "Optimal Histograms for Limiting Worst-Case

Error Propagation in the Size of Join Results", *ACM Transactions on Database Systems*, Volume 18, Número 4 (1993), páginas 709-748.

[Ioannidis e Poosala 1995] Y. E. Ioannidis e V. Poosala, "Balancing Histogram Optimality and Practicality for Query Result Size Estimation", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 233-244.

[Ioannidis *et al.* 1992] Y. E. Ioannidis, R. T. Ng, K. Shim e T. K. Sellis, "Parametric Query Optimization", *Proc. of the International Conf. on Very Large Databases* (1992), páginas 103-114.

[Jackson e Moulinier 2002] P. Jackson e I. Moulinier, *Natural Language Processing for Online Applications: Text Retrieval, Extraction, and Categorization*, John Benjamin (2002).

[Jagadish *et al.* 1993] H. V. Jagadish, A. Silberschatz e S. Sudarshan, "Recovering from Main-Memory Lapses", *Proc. of the International Conf. on Very Large Databases* (1993).

[Jagadish *et al.* 1994] H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz e S. Sudarshan, *Dali: A High Performance Main Memory Storage Manager* (1994).

[Jain e Dubes 1988] A. K. Jain e R. C. Dubes, *Algorithms for Clustering Data*, Prentice Hall (1988).

[Jensen *et al.* 1994] C. S. Jensen *et al.*, "A Consensus Glossary of Temporal Database Concepts", *ACM SIGMOD Record*, Volume 23, Número 1 (1994), páginas 52-64.

[Jensen *et al.* 1996] C. S. Jensen, R. T. Snodgrass e M. Soo, "Extending Existing Dependency Theory to Temporal Databases", *IEEE Transactions on Knowledge and Data Engineering*, Volume 8, Número 4 (1996), páginas 563-582.

[Jhingran *et al.* 1997] A. Jhingran, T. Malkemus e S. Padmanabhan, "Query Optimization in DB2 Parallel Edition", *Data Engineering Bulletin*, Volume 20, Número 2 (1997), páginas 27-34.

[Johnson 1999] T. Johnson, "Performance Measurements of Compressed Bitmap Indices", *Proc. of the International Conf. on Very Large Databases* (1999).

[Johnson e Shasha 1993] T. Johnson e D. Shasha, "The Performance of Concurrent B-Tree Algorithms", *ACM Transactions on Database Systems*, Volume 18, Número 1 (1993).

[Jones e Willet 1997] K. S. Jones e P. Willet(ed.), *Readings in Information Retrieval*, Morgan Kaufmann (1997).

[Jordan e Russell 2003] D. Jordan e C. Russell, *Java Data Objects*, O'Reilly (2003).

[Joshi 1991] A. Joshi, "Adaptive Locking Strategies in a Multi-Node Shared Data Model Environment", *Proc. of the International Conf. on Very Large Databases* (1991).

[Joshi *et al.* 1998] A. Joshi, W. Bridge, J. Loaiza e T. Lahiri, "Checkpointing in Oracle", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 665-668.

[Kanne e Moerkotte 2000] C.-C. Kanne e G. Moerkotte, "Efficient Storage of XML Data", *Proc. of the International Conf. on Data Engineering* (2000), página 198.

[Kapitskaia *et al.* 2000] O. Kapitskaia, R. T. Ng e D. Srivastava, "Evolution and Revolutions in LDAP Directory Caches", *Proc. of the International Conf. on Extending Database Technology* (2000), páginas 202-216.

[Katz *et al.* 2004] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy e P. Wadler, *XQuery from the Experts: A Guide to the W3C XML Query Language*, Addison Wesley (2004).

[Kaushik *et al.* 2004] R. Kaushik, R. Krishnamurthy, J. F. Naughton e R. Ramakrishnan, "On the Integration of Structure Indexes and Inverted Lists", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004).

[Kedem e Silberschatz 1979] Z. M. Kedem e A. Silberschatz, "Controlling Concurrency Using Locking Protocols", *Proc. of the Annual IEEE Symposium on Foundations of Computer Science* (1979), páginas 275-285.

[Kedem e Silberschatz 1983] Z. M. Kedem e A. Silberschatz, "Locking Protocols: From Exclusive to Shared Locks", *ACM Press*, Volume 30, Número 4 (1983), páginas 787-804.

[Kim 1982] W. Kim, "On Optimizing an SQL-like Nested Query", *ACM Transactions on Database Systems*, Volume 3, Número 3 (1982), páginas 443-469.

[Kim 1995] W. Kim, editor, *Modern Database Systems*, ACM Press/ Addison Wesley (1995).

[King 1981] J. J. King, "QUIST: A System for Semantic Query Optimization in Relational Data Bases", *Proc. of the International Conf. on Very Large Databases* (1981), páginas 510-517.

[King *et al.* 1991] R. P. King, N. Halim, H. Garcia-Molina e C. Polyzois, "Management of a Remote Backup Copy for Disaster Recovery,'" *ACM Transactions on Database Systems*, Volume 16, Número 2 (1991), páginas 338-368.

[Kitsuregawa e Ogawa 1990] M. Kitsuregawa e Y. Ogawa, "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Skew in the Super Database Computer", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 210-221.

[Kitsuregawa *et al.* 1983] M. Kitsuregawa, H. Tanaka e T. MotoOka, "Application of Hash to a Database Machine and its Architecture", *New Generation Computing*, Volume 1, Número 1 (1983), páginas 62-74.

[Kleinberg 1999] J. M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment", *journal of the ACM*, Volume 46, Número 5 (1999), páginas 604-632.

[Kleinrock 1975] L. Kleinrock, *Queuing Systems, Volume I: Theory*, John Wiley and Sons (1975).

[Kleinrock 1976] L. Kleinrock, *Queuing Systems, Volume 2: Computer Applications*, John Wiley and Sons (1976).

[Klug 1982] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions", *ACM Press*, Volume 29, Número 3 (1982), páginas 699-717.

[Knapp 1987] E. Knapp, "Deadlock Detection in Distributed Databases", *ACM Computing Survey*, Volume 19, Número 4 (1987).

[Knuth 1973] D. E. Knuth, *The Art of Computer Programming, Volume 3*, Addison Wesley, Sorting and Searching (1973).

[Kohavi e Provost 2001] R. Kohavi e F. Provost(ed.), *Applications of Data Mining to Electronic Commerce*, Kluwer Academic Publishers (2001).

[Konstan *et al.* 1997] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon e J. Riedl, "GroupLens: Applying Collaborative Filtering to Usenet News", *Communications of the ACM*, Volume 40, Número 3 (1997), páginas 77-87.

[Korth 1982] H. F. Korth, "Deadlock Freedom Using Edge Locks", *ACM Transactions on Database Systems*, Volume 7, Número 4 (1982), páginas 632-652.

[Korth 1983] H. F. Korth, "Locking Primitives in a Database System", *Journal of the ACM*, Volume 30, Número 1 (1983), páginas 55-79.

[Korth e Speegle 1994] H. F. Korth e G. Speegle, "Formal Aspects of Concurrency Control in Long Duration Transaction Systems Using the NT/PV Model", *ACM Transactions on Database Systems*, Volume 19, Número 3 (1994), páginas 492-535.

[Korth *et al.* 1990] H. F. Korth, E. Levy e A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions", *Proc. of the International Conf. on Very Large Databases* (1990).

[Krishnaprasad *et al.* 2004] M. Krishnaprasad, Z. Liu, A. Manikutty, J. W. Warner, V. Arora e S. Kotsovolos, "Query Rewrite for XML in Oracle XML DB", *Proc. of the International Conf. on Very Large Databases* (2004), páginas 1122-1133.

[Kung e Lehman 1980] H. T. Kung e P. L. Lehman, "Concurrent Manipulation of Binary Search Trees", *ACM Transactions on Database Systems*, Volume 5, Número 3 (1980), páginas 339-353.

[Kung e Robinson 1981] H. T. Kung e J. T. Robinson, "Optimistic Concurrency Control", *ACM Transactions on Database Systems*, Volume 6, Número 2 (1981), páginas 312-326.

[Labio *et al.* 1997] W. Labio, D. Quass e B. Adelberg, "Physical Database Design for Data Warehouses", *Proc. of the International Conf. on Data Engineering* (1997).

[Lahiri *et al.* 2001a] T. Lahiri, A. Ganesh, R. Weiss e A. Joshi, "Fast-Start: Quick Fault Recovery in Oracle", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2001).

[Lahiri *et al.* 2001b] T. Lahiri, V. Srihari, W. Chan, N. MacNaughton e S. Chandrasekaran, "Cache Fusion: Extending Shared-Disk Clusters with Shared Caches", *Proc. of the International Conf. on Very Large Databases* (2001), páginas 683-686.

[Lai e Wilkinson 1984] M. Y. Lai e W. K. Wilkinson, "Distributed Transaction Management in JASMIN", *Proc. of the International Conf. on Very Large Databases* (1984), páginas 466-472.

[Lam e Kuo 2001] K.-Y. Lam e T.-W. Kuo, editores, *Real-Time Database Systems*, Kluwer (2001).

[Lamb *et al.* 1991] C. Lamb, G. Landis, J. Orenstein e D. Weinreb, "The ObjectStore Database System", *Communications of the ACM*, Volume 34, Número 10 (1991), páginas 51-63.

[Lamport 1978] L. Lamport, "Time, Clocks e the Ordering of Events in a Distributed System", *Communications of the ACM*, Volume 21, Número 7 (1978), páginas 558-565.

[Lampson e Sturgis 1976] B. Lampson e H. Sturgis, "Crash Recovery in a Distributed Data Storage System", Technical report, Computer Science Laboratory, Xerox Palo Alto Research Center,Palo Alto (1976).

[Lanzelotte *et al.* 1993] R. Lanzelotte, P. Valduriez e M. Zar, "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces", *Proc. of the International Conf. on Very Large Databases* (1993).

[Larson e Yang 1985] P. Larson e H. Z. Yang, "Computing Queries from Derived Relations", *Proc. of the International Conf. on Very Large Databases* (1985), páginas 259-269.

[Lecluse *et al.* 1988] C. Lecluse, P. Richard e F. Velez, "O2: An Object-Oriented Data Model", *Proc. of the International Conf. on Very Large Databases* (1988), páginas 424-433.

[Lehman e Yao 1981] P. L. Lehman e S. B. Yao, "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems*, Volume 6, Número 4 (1981), páginas 650-670.

[Lehner *et al.* 2000] W. Lehner, R. Sidle, H. Pirahesh e R. Cochrane, "Maintenance of Automatic Summary Tables", *Proc. of the ACM SIG-MOD Conf. on Management of Data* (2000), páginas 512-513.

[Li e Mozes 2004] W. Li e A. Mozes, "Computing Frequent Itemsets Inside Oracle 10G", *Proc. of the International Conf. on Very Large Databases* (2004), páginas 1253-1256.

[Lin *et al.* 1994] E. T. Lin, E. R. Omiecinski e S. Yalamanchili, "Large Join Optimization on a Hypercube Multiprocessor", *IEEE Transactions on Knowledge and Data Engineering*, Volume 6, Número 2 (1994), páginas 304-315.

[Lindsay *et al.* 1980] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, G. R. Putzolu, I. L. Traiger e B. W. Wade. "Notes on Distributed Databases". *In* Draffen e Poole (ed.), *Distributed Data Bases*, páginas 247-284. Cambridge University Press, Cambridge, Inglaterra (1980).

[Litwin 1978] W. Litwin, "Virtual Hashing: A Dynamically Changing Hashing", *Proc. of the International Conf. on Very Large Databases* (1978), páginas 517-523.

[Litwin 1980] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing", *Proc. of the International Conf. on Very Large Databases* (1980).

[Litwin 1981] W. Litwin, "Trie Hashing", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), páginas 19-29.

[Lo e Ravishankar 1996] M.-L. Lo e C. V. Ravishankar, "Spatial Hash-Joins", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Loeb 1998] L. Loeb, *Secure Electronic Transactions: Introduction and Technical Reference*, Artech House (1998).

[Lomet 1981] D. G. Lomet, "Digital B-trees", *Proc. of the International Conf. on Very Large Databases* (1981), páginas 333-344.

[Lynch 1983] N. A. Lynch, "Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control", *ACM Transactions on Database Systems*, Volume 8, Número 4 (1983), páginas 484-502.

[Lynch e Merritt 1986] N. A. Lynch e M. Merritt, "Introduction to the Theory of Nested Transactions", *Proc. of the International Conf. on Database Theory* (1986).

[Lynch *et al.* 1988] N. A. Lynch, M. Merritt, W. Weihl e A. Fekete, "A Theory of Atomic Transactions", *Proc. of the International Conf. on Database Theory* (1988), páginas 41-71.

[Mackert e Lohman 1986] L. F. Mackert e G. M. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries", *Proc. of the International Conf. on Very Large Databases* (1986).

[Maier 1983] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville (1983).

[Martin *et al.* 1989] J. Martin, K. K. Chapman e J. Leben, *DB2, Concepts, Design, and Programming*, Prentice Hall (1989).

[Mattison 1996] R. Mattison, *Data Warehousing: Strategies, Technologies, and Techniques*, McGraw Hill (1996).

[McHugh e Widom 1999] J. McHugh e J. Widom, "Query Optimization for XML", *Proc. of the International Conf. on Very Large Databases* (1999), páginas 315-326.

[Mehrotra *et al.* 1991] S. Mehrotra, R. Rastogi, H. F. Korth e A. Silberschatz, "Non-Serializable Executions in Heterogeneous Distributed Database Systems", *Proc. of the International Conf. on Parallel and Distributed Information Systems* (1991).

[Mehrotra *et al.* 2001] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. . Korth e A. Silberschatz, "Overcoming Heterogeneity and Autonomy in Multidatabase Systems"., *Inf. Comput.*, Volume 167, Número 2 (2001), páginas 137-172.

[Melton 2002] J. Melton, *Advanced SQL:1999 - Understanding Object-Relational and Other Advanced Features*, Morgan Kaufmann (2002).

[Melton e Eisenberg 2000] J. Melton e A. Eisenberg, *Understanding SQL and Java Together : A Guide to SQL, JDBC, and Related Technologies*, Morgan Kaufmann (2000).

[Melton e Simon 1993] J. Melton e A. R. Simon, *Understanding The New SQL: A Complete Guide*, Morgan Kaufmann (1993).

[Melton e Simon 2001] J. Melton e A. R. Simon, *SQL:1999 Understanding Relational Language Components*, Morgan Kaufmann (2001).

[Menasce *et al.* 1980] D. A. Menasce, G. Popek e R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases", *ACM Transactions on Database Systems*, Volume 5, Número 2 (1980), páginas 103-138.

[Microsoft 1997] Microsoft, *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*, Microsoft Press (1997).

[Mistry *et al.* 2001] H. Mistry, P. Roy, S. Sudarshan e K. Ramamritham, "Materialized View Selection and Maintenance Using Multi-Query Optimization", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2001).

[Mitchell 1997] T. M. Mitchell, *Machine Learning*, McGraw Hill (1997).

[Mohan 1990a] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operations on B-Tree indexes", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 392-405.

[Mohan 1990b] C. Mohan, "Commit-LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 406-418.

[Mohan 1993] C. Mohan, "IBM's Relational Database Products: Features and Technologies", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[Mohan e Levine 1992] C. Mohan e F. Levine, "ARIES/IM:An Efficient and High-Concurrency Index Management Method Using Write-Ahead Logging", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1992).

[Mohan e Lindsay 1983] C. Mohan e B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", *Proc. of the ACM Symposium on Principles of Distributed Computing* (1983).

[Mohan e Narang 1991] C. Mohan e I. Narang, "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment", *Proc. of the International Conf. on Very Large Databases* (1991).

[Mohan e Narang 1992] C. Mohan e I. Narang, "Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment", *Proc. of the International Conf. on Extending Database Technology* (1992).

[Mohan e Narang 1994] C. Mohan e I. Narang, "ARIES/CSA: A Method for Database Recovery in Client-Server Architectures", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1994), páginas 55-66.

[Mohan *et al.* 1986] C. Mohan, B. Lindsay e R. Obermarck, "Transaction Management in the R* Distributed Database Management System", *ACM Transactions on Database Systems*, Volume 11, Número 4 (1986), páginas 378-396.

[Mohan *et al.* 1992] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh e P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, Volume 17, Número 1 (1992).

[Moss 1985] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge (1985).

[Moss 1987] J. E. B. Moss, "Log-Based Recovery for Nested Transactions", *Proc. of the International Conf. on Very Large Databases* (1987), páginas 427-432.

[MSDN: XML Developer Center] MSDN: XML Developer Center, "XML and the Database", http://msdn.microsoft.com/XML/Building-XML/XMLandDatabase/default.aspx.

[Murthy e Banerjee 2003] R. Murthy e S. Banerjee, "XML Schemas in Oracle XML DB", *Proc. of the International Conf. on Very Large Databases* (2003), páginas 1009-1018.

[Nakayama *et al.* 1984] T. Nakayama, M. Hirakawa e T. Ichikawa, "Architecture and Algorithm for Parallel Execution of a Join Operation", *Proc. of the International Conf. on Data Engineering* (1984).

[Ng e Han 1994] R. T. Ng e J. Han, "Efficient and Effective Clustering Methods for Spatial Data Mining", *Proc. of the International Conf. on Very Large Databases* (1994).

[Nyberg *et al.* 1995] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray e D. B. Lomet, "AlphaSort: A Cache-Sensitive Parallel External Sort", *VLDB Journal*, Volume 4, Número 4 (1995), páginas 603-627.

[O'Neil e O'Neil 2000] P. O'Neil e E. O'Neil, *Database Principles, Programming, Performance*, 2ª edição, Morgan Kaufmann (2000).

[O'Neil e Quass 1997] P. O'Neil e D. Quass, "Improved Query Performance with Variant Indexes", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1997).

[O'Neil *et al.* 2004] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller e N. Westbury, "ORD-PATHs: Insert-Friendly XML Node Labels", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), páginas 903-908.

[Orenstein 1982] J. A. Orenstein, "Multidimensional Tries Used for Associative Searching", *Information Processing Letters*, Volume 14, Número 4 (1982), páginas 150-157.

[Ozcan *et al.* 1997] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek e A. Dogac, "Dynamic Query Optimization in Multidatabases", *Data Engineering Bulletin*, Volume 20, Número 3 (1997), páginas 38-45.

[Ozden *et al.* 1994] B. Ozden, A. Biliris, R. Rastogi e A. Silberschatz, "A Low-cost Storage Server for a Movie on Demand Database", *Proc. of the International Conf. on Very Large Databases* (1994).

[Ozden *et al.* 1996a] B. Ozden, R. Rastogi, P. Shenoy e A. Silberschatz, "Fault-Tolerant Architectures for Continuous Media Servers", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Ozden *et al.* 1996b] B. Ozden, R. Rastogi e A. Silberschatz, "On the Design of a Low-Cost Video-on-Demand Storage System", *Multimedia Systems Journal*, Volume 4, Número 1 (1996), páginas 40-54.

[Ozsoyoglu e Snodgrass 1995] G. Ozsoyoglu e R. Snodgrass, "Temporal and Real-Time Databases: A Survey", *IEEE Transactions on Knowledge and Data Engineering*, Volume 7, Número 4 (1995), páginas 513-532.

[Ozsu e Valduriez 1999] T. Ozsu e P. Valduriez, *Principles of Distributed Database Systems*, 2ª edição, Prentice Hall (1999).

[Padmanabhan *et al.* 2003] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston e M. Huras, "Multi-Dimensional Clustering: A New Data Layout Scheme in DB2", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003), páginas 637-641.

[Pal *et al.* 2004] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis e V. Zolotov, "Indexing XML Data Stored in a Relational Database", *Proc. of the International Conf. on Very Large Databases* (2004), páginas 1134-1145.

[Pang *et al.* 1995] H.-H. Pang, M. J. Carey e M. Livny, "Multiclass Scheduling in Real-Time Database Systems", *IEEE Transactions on Knowledge and Data Engineering*, Volume 2, Número 4 (1995), páginas 533-551.

[Papadimitriou 1979] C. H. Papadimitriou, "The Serializability of Concurrent Database Updates", *Journal of the ACM*, Volume 26, Número 4 (1979), páginas 631-653.

[Papadimitriou 1986] C. H. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, Rockville (1986).

[Papadimitriou *et al.* 1977] C. H. Papadimitriou, P. A. Bernstein e J. B. Rothnie, "Some Computational Problems Related to Database Concurrency Control", *Proc. of the Conf. on Theoretical Computer Science* (1977), páginas 275-282.

[Papakonstantinou *et al.* 1996] Y. Papakonstantinou, A. Gupta e L. Haas, "Capabilities-Based Query Rewriting in Mediator Systems", *Proc. of the International Conf. on Parallel and Distributed Information Systems* (1996).

[Parker *et al.* 1983] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser e C. Kline, "Detection of Mutual Inconsistency in Distributed Systems", *IEEE Transactions on Software Engineering*, Volume 9, Número 3 (1983), páginas 240-246.

[Patel e DeWitt 1996] J. Patel e D. J. DeWitt, "Partition Based Spatial-Merge Join", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Patterson 2004] D. P. Patterson, "Latency Lags Bandwidth", *Communications of the ACM*, Volume 47, Número 10 (2004), páginas 71-75.

[Patterson *et al.* 1988] D. A. Patterson, G. Gibson e R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1988), páginas 109-116.

[Pellenkoft *et al.* 1997] A. Pellenkoft, C. A. Galindo-Legaria e M. Kersten, "The Complexity of Transformation-Based Join Enumeration", *Proc. of the International Conf. on Very Large Databases*, Athens, Greece (1997), páginas 306-315.

[Plessl998] V. Pless, *Introduction to the Theory of Error-Correcting Codes*, 3ª edição, John Wiley and Sons (1998).

[Poe 1995] V. Poe, *Building a Data Warehouse for Decision Support*, Prentice Hall (1995).

[Poess e Floyd 2000] M. Poess e C. Floyd, "New TPC Benchmarks for Decision Support and Web Commerce", *ACM SIGMOD Record*, Volume 29, Número 4 (2000).

[Poess e Potapov 2003] M. Poess e D. Potapov, "Data Compression in Oracle", *Proc. of the International Conf. on Very Large Databases* (2003), páginas 937-947.

[Polyzois e Garcia-Molina 1994] C. Polyzois e H. Garcia-Molina, "Evaluation of Remote Backup Algorithms for Transaction-Processing Systems", *ACM Transactions on Database Systems*, Volume 19, Número 3 (1994), páginas 423-449.

[Poosala *et al.* 1996] V. Poosala, Y. E. Ioannidis, P. J. Haas e E. J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 294-305.

[Popek *et al.* 1981] G. J. Popek, B. J. Walker, J. M. Chow, D. Edwards, C. Kline, G. Rudisin e G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System", *Proc. of the Eighth Symposium on Operating System Principles* (1981), páginas 169-177.

[Pu *et al.* 1988] C. Pu, G. Kaiser e N. Hutchinson, "Split-Transactions for Open-Ended Activities", *Proc. of the International Conf. on Very Large Databases* (1988), páginas 26-37.

[Rahm 1993] E. Rahm, "Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems", *ACM Transactions on Database Systems*, Volume 8, Número 2 (1993).

[Ramakrishna e Larson 1989] M. V. Ramakrishna e P. Larson, "File Organization Using Composite Perfect Hashing", *ACM Transactions on Database Systems*, Volume 14, Número 2 (1989), páginas 231-263.

[Ramakrishnan e Gehrke 2002] R. Ramakrishnan e J. Gehrke, *Database Management Systems*, 3ª edição, McGraw Hill (2002).

[Ramakrishnan e Ullman 1995] R. Ramakrishnan e J. D. Ullman, "A Survey of Deductive Database Systems", *Journal of Logic Programming*, Volume 23, Número 2 (1995), páginas 125-149.

[Ramakrishnan *et al.* 1992] R. Ramakrishnan, D. Srivastava e S. Sudarshan, *Controlling the Search in Bottom-up Evaluation* (1992).

[Ramesh *et al.* 1989] R. Ramesh, A. J. G. Babu e J. P. Kincaid, "Index Optimization: Theory and Experimental Results", *ACM Transactions on Database Systems*, Volume 14, Número 1 (1989), páginas 41-74.

[Rangan *et al.* 1992] P. V. Rangan, H. M. Vin e S. Ramanathan, "Designing an On-Demand Multimedia Service", *Communications Magazine*, Volume 1, Número 1 (1992), páginas 56-64.

[Rao e Ross 2000] J. Rao e K. A. Ross, "Making B+-Trees Cache Conscious in Main Memory", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000), páginas 475-486.

[Rathi *et al.* 1990] A. Rathi, H. Lu e G. E. Hedrick, "Performance Comparison of Extendable Hashing and Linear Hashing Techniques", *Proc. ACM SIGSmall/PC Symposium on Small Systems* (1990), páginas 178-185.

[Reed 1978] D. Reed, *Naming and Synchronization in a Decentralized Computer System*. ????Tese de PhD ?????thesis, Department of Electrical Engineering, MIT, Cambridge (1978).

[Reed 1983] D. Reed, "Implementing Atomic Actions on Decentralized Data", *Transactions on Computer Systems*, Volume 1, Número 1 (1983), páginas 3-23.

[Revesz 2002] P. Revesz, *Introduction to Constraint Databases*, Springer Verlag (2002).

[Richardson *et al.* 1987] J. Richardson, H. Lu e K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987).

[Rivest 1976] R. L. Rivest, "Partial Match Retrieval Via the Method of Superimposed Codes", *SIAM Journal of Computing*, Volume 5, Número 1 (1976), páginas 19-50.

[Rizvi *et al.* 2004] S. Rizvi, A. Mendelzon, S. Sudarshan e P. Roy, "Extending Query Rewriting Techniques for Fine-Grained Access Control", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004).

[Robinson 1981] J. Robinson, "The k-d-B Tree: A Search Structure for Large Multidimensional Indexes", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), páginas 10-18.

[Roos 2002] R. M. Roos, *Java Data Objects*, Pearson Education (2002).

[Rosch 2003] W. L. Rosch, *The Winn L. Rosch Hardware Bible*, 6ª edição, Que (2003).

[Rosenblum e Ousterhout 1991] M. Rosenblum e J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", *Proc. of the International Conf. on Architectural Support for Programming Languages and Operating Systems* (1991), páginas 1-15.

[Rosenthal e Reiner 1984] A. Rosenthal e D. Reiner, "Extending the Algebraic Framework of Query Processing to Handle Outerjoins", *Proc. of the International Conf. on Very Large Databases* (1984), páginas 334-343.

[Ross 1990] K. A. Ross, "Modular Stratification and Magic Sets for DATALOG Programs with Negation", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990).

[Ross 1999] S. M. Ross, *Introduction to Probability and Statistics for Engineers and Scientists*, Harcourt/Academic Press (1999).

[Ross e Srivastava 1997] K. A. Ross e D. Srivastava, "Fast Computation of Sparse Datacubes", *Proc. of the International Conf. on Very Large Databases* (1997), páginas 116-125.

[Ross *et al.* 1996] K. Ross, D. Srivastava e S. Sudarshan, "Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Rothermel e Mohan 1989] K. Rothermel e C. Mohan, "ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions", *Proc. of the International Conf. on Very Large Databases* (1989), páginas 337-346.

[Roy *et al.* 2000] P. Roy, S. Seshadri, S. Sudarshan e S. Bhobhe, "Efficient and Extensible Algorithms for Multi-Query Optimization", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000).

[Rusinkiewicz e Sheth 1995] M. Rusinkiewicz e A. Sheth, "Specification and Execution of Transactional Workflows", Em *Kim [1995]*, páginas 592-620 (1995).

[Rustin 1972] R. Rustin, *Data Base Systems*, Prentice Hall (1972).

[Rys 2001] M. Rys, "Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems", *Proc. of the International Conf. on Data Engineering* (2001), páginas 465-472.

[Rys 2003] M. Rys, "XQuery and Relational Database Systems", Em H. Katz, editor, *XQuery From the Experts*, páginas 353-391. Addison Wesley (2003).

[Rys 2004] M. Rys, "What's New in FOR XML in Microsoft SQL Server 2005", http://msdn.microsoft.com/library/en-us/dnsql90/html/forxml2k5.asp(2004).

[Sagiv e Yannakakis 1981] Y. Sagiv e M. Yan-nakakis, "Equivalence among Relational Expressions with the Union and Difference Operators", *Proc. of the ACM SIGMOD Conf. on Management of Data*, Volume 27, Número 4 (1981).

[Salem e Garcia-Molina 1986] K. Salem e H. Garcia-Molina, "Disk Striping", *Proc. of the International Conf. on Data Engineering* (1986), páginas 336-342.

[Salem *et al.* 1994] K. Salem, H. Garcia-Molina e J. Sands, "Altruistic Locking", *ACM Transactions on Database Systems*, Volume 19, Número 1 (1994), páginas 117-165.

[Salton 1989] G. Salton, *Automatic Text Processing*, Addison Wesley (1989).

[Samet 1990] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison Wesley (1990).

[Samet 1995a] H. Samet, "General Research Issues in Multimedia Database Systems", *ACM Computing Survey*, Volume 27, Número 4 (1995), páginas 630-632.

[Samet 1995b] H. Samet, "Spatial Data Structures", Em *Kim [1995]*, páginas 361-385 (1995).

[Samet e Aref 1995] H. Samet e W. Aref, "Spatial Data Models and Query Processing", Em *Kim [1995]*, páginas 338-360 (1995).

[Sanders 1998] R. E. Sanders, *ODBC 3.5 Developer's Guide*, McGraw Hill (1998).

[Sanders 2000] R. E. Sanders, *DB2 Universal Database SQL Developer's Guide*, McGraw Hill (2000).

[Sarawagi 2000] S. Sarawagi, "User-Adaptive Exploration of Multidimensional Data", *Proc. of the International Conf. on Very Large Databases* (2000), páginas 307-316.

[Sarawagi *et al.* 2002] S. Sarawagi, A. Bhamidi-paty, A. Kirpal e C. Mouli, "ALIAS: An Active Learning Led Interactive Deduplication System", *Proc. of the International Conf. on Very Large Databases* (2002), páginas 1103-1106.

[Schlageter 1981] G. Schlageter, "Optimistic Methods for Concurrency Control in Distributed Database Systems", *Proc. of the International Conf. on Very Large Databases* (1981), páginas 125-130.

[Schneider 1982] H. J. Schneider, *Distributed Data Bases* (1982).

[Schneider e DeWitt 1989] D. Schneider e D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989).

[Schning 2001] H. Schning, "Tamino - A DBMS Designed for XML", *Proc. of the International Conf. on Data Engineering* (2001), páginas 149-154.

[Selinger e Adiba 1980] P. G. Selinger e M. E. Adiba, "Access Path Selection in Distributed Database Management Systems", Technical Report RJ2338, IBM Research Laboratory, San Jose (1980).

[Selinger *et al.* 1979] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie e T. G. Price, "Access Path Selection in a Relational Database System", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979), páginas 23-34.

[Sellis1988] T. K. Sellis, "Multiple Query Optimization", *ACM Transactions on Database Systems*, Volume 13, Número 1 (1988), páginas 23-52.

[Sellis *et al.* 1987] T. K. Sellis, N. Roussopoulos e C. Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects", *Proc. of the International Conf. on Very Large Databases* (1987), páginas 507-518.

[Seshadri *et al.* 1996] P. Seshadri, H. Pirahesh e T. Y. C. Leung, "Complex Query Decorrelation", *Proc. of the International Conf. on Data Engineering* (1996), páginas 450-458.

[Shafer *et al.* 1996] J. C. Shafer, R. Agrawal e M. Mehta, "SPRINT: A Scalable Parallel Classifier for Data Mining", *Proc. of the International Conf. on Very Large Databases* (1996).

[Shanmugasundaram *et al.* 1999] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt e J. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities", *Proc. of the International Conf. on Very Large Databases* (1999).

[Shapiro 1986] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems*, Volume 11, Número 3 (1986), páginas 239-264.

[Shasha e Bonnet 2002] D. Shasha e P. Bonnet, *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann (2002).

[Shasha *et al.* 1995] D. Shasha, F. Llirabat, E. Simon e P. Valduriez, "Transaction Chopping: Algorithms and Performance Studies", *ACM Transactions on Database Systems*, Volume 20, Número 3 (1995), páginas 325-363.

[Shatdal e Naughton 1993] A. Shatdal e J. Naughton, "Using Shared Virtual Memory for Parallel Join Processing", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[Silberschatz 1982] A. Silberschatz, "A Multi-Version Concurrency Control Scheme With No Rollbacks", *Proc. of the ACM Symposium on Principles of Distributed Computing* (1982), páginas 216-223.

[Silberschatz e Kedem 1980] A. Silberschatz e Z. Kedem, "Consistency in Hierarchical Database Systems", *Journal of the ACM*, Volume 27, Número 1 (1980), páginas 72-80.

[Silberschatz *et al.* 1990] A. Silberschatz, M. R. Stonebraker e J. D. Ullman, "Database Systems: Achievements and Opportunities", *ACM SIGMOD Record*, Volume 19, Número 4 (1990).

[Silberschatz *et al.* 1996] A. Silberschatz, M. Stonebraker e J. Ullman, "Database Research: Achievements and Opportunities into the 21st Century", Technical Report CS-TR-96-1563, Department of Computer Science, Stanford University, Stanford (1996).

[Silberschatz *et al.* 2001] A. Silberschatz, P. B. Galvin e G. Gagne, *Operating System Concepts*, 6ª edição, John Wiley and Sons (2001).

[Simmen *et al.* 1996] D. Simmen, E. Shekita e T. Malkemus, "Fundamental Techniques for Order Optimization", *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada (1996), páginas 57-67.

[Skeen 1981] D. Skeen, "Non-blocking Commit Protocols", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), páginas 133-142.

[Soderland 1999] S. Soderland, "Learning Information Extraction Rules for Semi-structured and Free Text", *Machine Learning*, Volume 34, Número 1-3 (1999), páginas 233-272.

[Soo 1991] M. Soo, "Bibliography on Temporal Databases", *ACM SIGMOD Record*, Volume 20, Número 1 (1991), páginas 14-23.

[Spector e Schwarz 1983] A. Z. Spector e P. M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing", *Operating Systems Review*, Volume 17, Número 2 (1983), páginas 18-35.

[SQL/XML 2004] SQL/XML. "ISO/IEC 9075-14:2003, Information Technology: Database languages: SQL Part 14: XML-Related Specifications (SQL/XML)" (2004).

[Srikant e Agrawal 1996a] R. Srikant e R. Agrawal, "Mining Quantitative Association Rules in Large Relational Tables", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Srikant e Agrawal 1996b] R. Srikant e R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements", *Proc. of the International Conf. on Extending Database Technology* (1996), páginas 3-17.

[Srinivasan *et al.* 2000a] J. Srinivasan, S. Das, C. Freiwald, E. I. Chong, M. Jagannath, A. Yalamanchi, R. Krishnan, A.-T. Tran, S. DeFazio e J. Banerjee, "Oracle 8i Index-

Organized Table and Its Application to New Domains", *Proc. of the International Conf. on Very Large Databases* (2000), páginas 285-296.

[Srinivasan *et al.* 2000b] J. Srinivasan, R. Murthy, S. Sundara, N. Agarwal e S. DeFazio, "Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle 8i", *Proc. of the International Conf. on Data Engineering* (2000), páginas 91-100.

[Stam e Snodgrass 1988] R. Stam e R. Snodgrass, "A Bibliography on Temporal Databases", *IEEE Transactions on Knowledge and Data Engineering*, Volume 7, Número 4 (1988), páginas 231-239.

[Stinson 2002] B. Stinson, *PostgreSQL Essential Reference*, New Riders (2002).

[Stonebraker 1986] M. Stonebraker, "Inclusion of New Types in Relational Database Systems", *Proc. of the International Conf. on Data Engineering* (1986), páginas 262-269.

[Stonebraker e Rowe 1986] M. Stonebraker e L. Rowe, "The Design of POSTGRES", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1986).

[Stonebraker *et al.* 1989] M. Stonebraker, P. Aoki e M. Seltzer, "Parallelism in XPRS", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989).

[Stonebraker *et al.* 1990] M. Stonebraker, A. Jhingran, J. Goh e S. Potamianos, "On Rules, Procedure, Caching and Views in Database Systems", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), páginas 281-290.

[Stuart *et al.* 1984] D. G. Stuart, G. Buckley e A. Silberschatz, "A Centralized Deadlock Detection Algorithm", Technical report, Department of Computer Sciences, University of Texas, Austin (1984).

[Sudarshan e Ramakrishnan 1991] S. Sudarshan e R. Ramakrishnan, "Aggregation and Relevance in Deductive Databases", *Proc. of the International Conf. on Very Large Databases* (1991).

[Tanenbaum 2002] A. S. Tanenbaum, *Computer Networks*, 4ª edição, Prentice Hall (2002).

[Tansel *et al.* 1993] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev e R. Snodgrass, *Temporal Databases: Theory, Design and Implementation*, Benjamin Cummings, Redwood City (1993).

[Teorey *et al.* 1986] T. J. Teorey, D. Yang e J. P. Fry, "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model", *ACM Computing Survey*, Volume 18, Número 2 (1986), páginas 197-222.

[Thalheim 2000] B. Thalheim, *Entity-Relationship Modeling : Foundations of Database Technology*, Springer Verlag (2000).

[Thomas 1979] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control", *ACM Transactions on Database Systems*, Volume 4, Número 2 (1979), páginas 180-219.

[Thomas 1996] S. A. Thomas, *IPng and the TCP/IP Protocols: Implementing the Next Generation Internet*, John Wiley and Sons (1996).

[Traiger *et al.* 1982] I. L. Traiger, J. N. Gray, C. A. Galtieri e B. G. Lindsay, "Transactions and Consistency in Distributed Database Management Systems", *ACM Transactions on Database Systems*, Volume 7, Número 3 (1982).

[Tremblay e Sorenson 1985] J. P. Tremblay e P. G. Sorenson, *The Theory and Practice of Compiler Writing*, McGraw Hill (1985).

[Tsukuda *et al.* 1992] S. Tsukuda, M. Nakano, M. Kitsuregawa e M. Takagi, "Parallel Hash Join on Shared-Everything Multiprocessor", *Proc. of the International Conf. on Data Engineering* (1992).

[Tyagi *et al.* 2003] S. Tyagi, M. Vorburger, K. McCammon e H. Bobzin, *Core Java Data Objects*, Prentice Hall (2003).

[Ullman 1988] J. D. Ullman, *Principles of Database and Knowledge-base Systems, Volume 1*, Computer Science Press, Rockville (1988).

[Ullman 1989] J. D. Ullman, *Principles of Database and Knowledge-base Systems, Volume 2*, Computer Science Press, Rockville (1989).

[Umar 1997] A. Umar, *Application (Re)Engineering : Building Web-Based Applications and Dealing With Legacies*, Prentice Hall (1997).

[UniSQL 1991] *UniSQL/X Database Management System User's Manual: Release 1.2*. UniSQL, Inc. (1991).

[Verhofstad 1978] J. S. M. Verhofstad, "Recovery Techniques for Database Systems", *ACM Computing Survey*, Volume 10, Número 2 (1978), páginas 167-195.

[Vista 1998] D. Vista, "Integration of Incremental View Maintenance into Query Optimizers", *Proc. of the International Conf. on Extending Database Technology* (1998).

[Vitter 2001] J. S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data", *ACM Computing Surveys*, Volume 33, (2001), páginas 209-271.

[Wachter e Reuter 1992] H. Wachter e A. Reuter. "The ConTract Model", Em A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann (1992).

[Walsh *et al.* 2004] N. Walsh *et al.* "XQuery 1.0 and XPath 2.0 Data Model". http://www.w3.org/TR/xpath-datamodel/. currently a W3C Working Draft (2004).

[Walton *et al.* 1991] C. Walton, A. Dale e R. Jenevein, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", *Proc. of the International Conf. on Very Large Databases* (1991).

[Weihl e Liskov 1990] W. Weihl e B. Liskov. "Implementation of Resilient, Atomic Data Types", Em *Zdonik and Maier [1990]*, páginas 332-344 (1990).

[Weikum 1991] G. Weikum, "Principles and Realization Strategies of Multilevel Transaction Management", *ACM Transactions on Database Systems*, Volume 16, Número 1 (1991).

[Weikum e Schek 1984] G. Weikum e H. J. Schek, "Architectural Issues of Transaction Management in Multi-Level Systems", *Proc. of the International Conf. on Very Large Databases* (1984), páginas 454-465.

[Weikum *et al.* 1990] G. Weikum, C. Hasse, P. Broessler e P. Muth, "Multi-Level Recovery", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), páginas 109-123.

[Weltman e Dahbura 2000] R. Weltman e T. Dahbura, *LDAP Programming with Java*, Addison Wesley (2000).

[Wilschut *et al.* 1995] A. N. Wilschut, J. Flokstra e P. M. Apers, "Parallel Evaluation of Multi-Join Queues", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 115-126.

[Wipfler 1987] A. J. Wipfler, *CICS: Application Development and Programming*, Macmillan Publishing, New York (1987).

[Witkowski *et al.* 2003a] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng e S. Subramanian, "Spreadsheets in RDBMS for OLAP", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003), páginas 52-63.

[Witkowski *et al.* 2003b] A. Witkowski, S. Bellamkonda, T. Bozkaya, N. Folkert, A. Gupta, L. Sheng e S. Subramanian, "Business Modelling Using SQL Spreadsheets", *Proc. of the International Conf. on Very Large Databases* (2003), páginas 1117-1120.

[Witten e Frank 1999] I. H. Witten e E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann (1999).

[Witten *et al.* 1999] I. H. Witten, A. Moffat e T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*, Morgan Kaufmann (1999).

[Wolf 1991] J. Wolf, "An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew", *Proc. of the International Conf. on Data Engineering* (1991).

[Wong 1977] E. Wong, "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", *Proc. of the Berkeley Workshop on Distributed Data Management and Computer Networks* (1977), páginas 217-235.

[Wu e Buchmann 1998] M. Wu e A. Buchmann, "Encoded Bitmap Indexing for Data Warehouses", *Proc. of the International Conf. on Data Engineering* (1998).

[Wu *et al.* 2003] Y. Wu, J. M. Patel e H. V. Jagadish, "Structural Join Order Selection for XML Query Optimization", *Proc. of the International Conf. on Data Engineering* (2003).

[X/Open 1991] *X/Open Snapshot: X/Open DTP: XA Interface*. X/Open Company, Ltd. (1991).

[Yan e Larson 1995] W. P. Yan e P. A. Larson, "Eager Aggregation and Lazy Aggregation", *Proc. of the International Conf. on Very Large Databases*, Zurich (1995).

[Yannakakis *et al.* 1979] M. Yannakakis, C. H. Papadimitriou e H. T. Kung, "Locking Protocols: Safety and Freedom from Deadlock", *Proc. of the IEEE Symposium on the Foundations of Computer Science* (1979), páginas 286-297.

[Zaharioudakis *et al.* 2000] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh e M. Urata, "Answering Complex SQL Queries using Automatic Summary Tables", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000), páginas 105-116.

[Zaniolo *et al.* 1997] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, R. Zicari e V. S. Subrahmanian, *Advanced Database Systems*, Morgan Kaufmann (1997).

[Zdonik e Maier 1990] S. Zdonik e D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann (1990).

[Zeller e Gray 1990] H. Zeller e J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 186-197.

[Zhang *et al.* 1996] T. Zhang, R. Ramakrishnan e M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 103-114.

[Zhou e Ross 2004] J. Zhou e K. A. Ross, "Buffering Database Operations for Enhanced Instruction Cache Performance", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), páginas 191-202.

[Zhuge *et al.* 1995] Y. Zhuge, H. Garcia-Molina, J. Hammer e J. Widom, "View Maintenance in a Warehousing Environment", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 316-327.

[Zikopoulos *et al.* 2000] P. Zikopoulos, R. Melnyk e L. Logomirski, *DB2 for Dummies*, Hungry Minds, Inc. (2000).

[Zikopoulos *et al.* 2004] P. Zikopoulos, G. Baklarz, D. deRoos e R. B. Melnyk, *DB2 Version 8: The Official Guide*, IBM Press (2004).

[Zilio *et al.* 2004] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C Garcia-Arellano e S. Fadden, "DB2 Design Advisor: Integrated Automatic Physical Database Design", *Proc. of the International Conf. on Very Large Databases* (2004).

[Zloof 1977] M. M. Zloof, "Query-by-Example: A Data Base Language", *IBM Systems Journal*, Volume 16, Número 4 (1977), páginas 324-343.

Índice

B

F

G

J

K

L

Q

R

S

X

Y

Z

A quinta edição do livro *Sistema de Banco de Dados* de Silberschatz, Korth e Sudarshan é uma referência essencial para aqueles que precisam ter uma boa base teórica e prática em banco de dados. Além dos conceitos fundamentais da área, são apresentados tópicos relacionados, tais como: desenvolvimento para Internet, banco de dados orientados a objetos, XML, data mining, processamento paralelo e distribuído. O livro também inclui vários comentários sobre as mais recentes padronizações da linguagem SQL além de abordar em estudos de casos as características dos SGDBs mais utilizados atualmente.

Luiz Fernando Pereira de Souza
Mestre em Engenharia de Sistemas pela UFRJ
Professor e Analista de Sistema do NCE/UFRJ

ELSEVIER
CAMPUS
Uma empresa Elsevier
www.campus.com.br