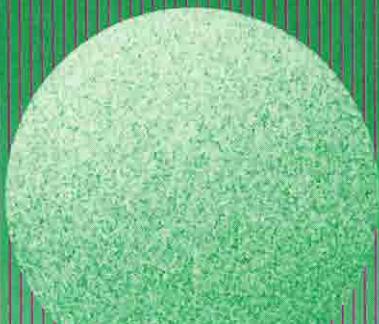


7.^a
Reimpressão

C.J. DATE

Introdução a Sistemas de
**BANCOS DE
DADOS**



Editora Campus

TÍTULOS DE INTERESSE CORRELATO

BANCO DE DADOS

BANCO DE DADOS: Fundamentos — C.J. Date

GERÊNCIA DE BASES DE DADOS PARA MICROCOMPUTADORES — E.G.

Brooker

ORGANIZAÇÃO DE BANCOS DE DADOS — A.L. Furtado e C.S. Santos

PRINCÍPIOS DE SISTEMAS DE GERÊNCIA DE BANCOS DE DADOS

DISTRIBUÍDOS — M.A. Casanova e A.V. Moura

BANCO DE DADOS: Tópicos Avançados — C.J. Date

GUIA PARA O DB2 — C.J. Date

GUIA PARA O PADRÃO SQL — C.J. Date

Solicite nosso catálogo completo.

Procure nossas publicações nas boas livrarias ou comunique-se diretamente com:

EDITORAS CAMPUS LTDA.

Qualidade internacional a serviço do autor e do leitor nacional.

Rua Barão de Itapagipe, 55 Rio Comprido

Tel. PABX (021) 293-6443 Telex (021) 32606 EDCP BR

20261 Rio de Janeiro RJ Brasil

Endereço Telegráfico: CAMPUSRIO

Introdução a Sistemas de **BANCOS DE** **DADOS**



C.J.DATE

IBM Corporation

7.^a Reimpressão

Introdução *a Sistemas de* **BANCOS DE** **DADOS**

TRADUÇÃO

Hélio Auro Gouveia



TRADUÇÃO INTEGRAL E AUTORIZADA DO ORIGINAL

Editora Campus Ltda.

Do original:
An introduction to Database Systems by C.J. Date.

Tradução autorizada da edição inglesa © 1981, 1977 e 1975 by Addison-Wesley Publishing Company, Inc.
Esta tradução é publicada e vendida com permissão de Addison-Wesley Publishing Company Inc., a
proprietária de todos os direitos de publicação e venda da mesma.

© 1984, Editora Campus Ltda.
Tradução autorizada da edição inglesa.
7^ª Reimpressão, 1989.

Todos os direitos reservados e protegidos pela Lei 5988 de 14/12/73.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou
transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou
qualsquer outros.

Todo o esforço foi feito para fornecer a mais completa e adequada informação.
Contudo a editora e o(s) autor(es) não assumem responsabilidade pelos resultados e uso da informação
fornecida. Recomendamos aos leitores testar a informação antes de sua efetiva utilização.

Capa
Otavio Studart

Projeto Gráfico, Composição e Revisão
Editora Campus Ltda.

Qualidade internacional a serviço do autor e do leitor nacional.

Rua Barão de Itapagipe 55 Rio Comprido
Telefone: (021) 293 6443 Telex: (021) 32606 EDCP BR
20261 Rio de Janeiro RJ Brasil
Endereço Telegráfico: CAMPUSRIO

ISBN 85-7001-392-2

(Edição original: ISBN 0-201-14471-9, Addison-Wesley Publishing Company, Inc., USA.)

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

Date, C.J.

D257 Introdução a sistemas de bancos de dados / C.J. Date; tradução de Hélio Auro Gouveia. — Rio de Janeiro: Campus, 1984.

Tradução de: An introduction to database systems.
Bibliografia.
ISBN 85-7001-392-2

1. Banco de dados — Organização. I. Título.

CDD — 001.6425
CDU — 061.68

86-0978

93 92 91 90 89 9 8 7 6 5 4 3 2 1

O Autor

C. J. Date é um Programador Assessor da IBM General Products Division, San Jose, California.

Ao se formar em matemática na Cambridge University (Inglaterra) em 1962, foi trabalhar na Leo Computers Ltd, Londres, como programador e instrutor de programação. Transferiu-se para a IBM em 1967 para, como instrutor, auxiliar no desenvolvimento e ministrar um programa amplo de treinamento em fundamentos de sistemas de computadores, linguagem assembler do Sistema/360, e PL/I. Subseqüentemente, ajudou a organizar o "IBM European Laboratories Integrated Professional Training Program (ELIPT)", um amplo esquema cooperativo de educação voltado para os profissionais de computação dos laboratórios de desenvolvimento IBM na Áustria, Inglaterra, França, Alemanha, Holanda e Suécia. Este trabalho incluiu o desenvolvimento e o ensino de diversos novos cursos, cobrindo tópicos como técnicas de programação de sistemas e Operating System/360 (tanto a nível do usuário quanto da codificação interna do OS/360).

Em 1970, o Sr. Date trabalhou em um projeto de linguagem de banco de dados na IBM (Inglaterra). Desde aquela época, tem se mantido mais ou menos continuamente ativo no campo dos bancos de dados, tanto na IBM quanto fora dela. Em particular, projetou e ministrou um curso muito bem-sucedido de conceitos de banco de dados no programa IBM ELIPT previamente mencionado. Este livro tirou grande proveito da experiência do autor oriunda do ensino desse curso. O Sr. Date é também responsável pelo projeto de uma linguagem de programação proposta para bancos de dados conhecida como UDL (Unified Database Language – Linguagem unificada para bancos de dados). Além disso o autor lecionou extensamente sobre tópicos de bancos de dados – particularmente sobre bancos de dados relacionais – tanto nos Estados Unidos como em muitos outros países. É membro da ACM e do "ACM Special Interest Group on Management of Data (SIGMOD)". Envolveu-se ativamente durante algum tempo em um grupo de trabalho da "British Computer Society" na área de bancos de dados relacionais. É autor/co-autor de diversos artigos técnicos.

Prefácio da Primeira Edição

Os computadores causaram um impacto considerável em muitos aspectos da nossa sociedade. Medicina, justiça, governo, bancos, educação, transporte, planejamento — estas são somente algumas das áreas onde os computadores já desempenham um papel altamente significativo. Podemos esperar para os próximos anos um grande incremento na faixa de aplicação dos computadores e um correspondente aumento do efeito que os computadores terão sobre a nossa vida diária. As duas áreas de tecnologia de computadores que tornarão possíveis as novas aplicações. — em muitos casos elas são sem dúvida fundamentais — são telecomunicações e os bancos de dados integrados.

Nos anos vindouros, então, os sistemas de bancos de dados tornar-se-ão cada vez mais difundidos e importantes. Presentemente, entretanto, representam um campo novo e relativamente inexplorado, muito embora o número de sistemas instalados ou em desenvolvimento esteja crescendo a uma velocidade considerável. Isto gera uma necessidade real de que exista um bom livro texto básico que cubra os conceitos fundamentais de tais sistemas de uma forma clara e concisa. Este livro representa uma tentativa de suprir essa necessidade.

Partiu-se do princípio de que o leitor está profissionalmente interessado em algum aspecto de processamento de dados. Ele ou ela pode ser, por exemplo, analista ou projetista de sistemas, programador(a) de aplicações, programador(a) de sistemas, estudante de curso universitário ou similar em ciência da computação, ou professor de algum desses cursos. (O livro está de fato baseado em um curso intensivo sobre o assunto que o autor vem lecionando ao *staff* profissional interno da IBM há bastante tempo.) De forma genérica, espera-se que o leitor possua uma noção razoável sobre a capacidade de um moderno sistema de computadores, particularmente sobre os dispositivos de manuseio de arquivos de tal sistema. Ele ou ela deve também conhecer pelo menos uma linguagem de programação de alto nível. Como estes pré-requisitos não são particularmente exigentes, estou certo de que o livro se mostrará adequado como um texto introdutório para quem estiver interessado no uso ou implementação de um sistema de banco de dados, ou para quem deseja simplesmente ampliar o conhecimento geral no campo da ciência da computação.

O livro está dividido em seis grandes partes:*

- 1 – Arquitetura de Sistemas de Bancos de Dados
- 2 – A Abordagem Relacional
- 3 – A Abordagem Hierárquica
- 4 – A Abordagem de Rede
- 5 – Segurança e Integridade
- 6 – Revisão, Análises e Comparações

Por sua vez, cada parte está subdividida em um certo número de capítulos. A Parte 1 fornece uma introdução geral aos conceitos de um sistema de bancos de dados, esboçando em particular três abordagens distintas para o projeto desse sistema, a saber, relacional, hierárquico e de rede. A Parte 2 então examina a abordagem relacional de forma bastante detalhada; a Parte 3 desempenha a mesma função para a abordagem hierárquica; e a Parte 4 faz o mesmo para a abordagem de rede. A Parte 5 apresenta uma discussão sobre os problemas de segurança e integridade em um sistema de bancos de dados. A Parte 6 reúne alguns dos temas mais importantes introduzidos anteriormente no livro e os estuda com mais alguma profundidade.

A estrutura que acabou de ser definida precisa de alguma justificativa. Como explicado, a Parte 2 preocupa-se com a abordagem relacional. De fato, ela está extensamente dedicada à exposição das idéias do Dr. E.F. Codd, reconhecida autoridade no campo de bancos de dados relacionais. A bem da verdade, no entanto, é necessário esclarecer que a maioria dos sistemas comerciais correntemente disponíveis (1974) está baseada em uma das outras duas abordagens. Por que, então, a ênfase na abordagem relacional? Há pelo menos duas respostas a esta questão. 1. A abordagem relacional pode ser vista como a base de uma teoria de dados; assim sendo, ela fornece um excelente embasamento para se entender e comparar as duas outras abordagens, e um meio conveniente ou medida para comparação em relação à qual pode ser julgado qualquer sistema existente. A solidez e a continuidade dessa teoria fariam dela um veículo ideal para finalidades didáticas, mesmo que não tivesse outras vantagens. 2. O fato de a maioria dos sistemas não ser relacional pode ser considerado como uma consequência natural da forma como a própria tecnologia da computação foi desenvolvida. A capacidade comparativamente pequena e o grande tempo para acesso dos antigos dispositivos de acesso direto, a ênfase tradicional no método sequencial tal como fita ou cartão, a limitação na quantidade de memória disponível no próprio computador – estas considerações e outras semelhantes tiveram repercussão significativa no projeto original da maioria dos antigos sistemas. Com técnicas e *hardware* modernos, entretanto, parece ser possível projetar e construir um sistema sem as restrições dos projetos anteriores. Para ser mais específico, muitas autoridades acreditam hoje que o futuro verá a implementação de um ou mais sistemas de grande porte baseados na abordagem relacional. (Desde que isto foi escrito, de fato, começaram a surgir no cenário alguns sistemas comerciais incorporando conceitos relacionais.)

Destas observações o leitor concluirá, corretamente, que o texto está algo polarizado a favor da abordagem relacional. Obviamente o autor acredita que tal polarização se justifica; mas seria desonesto não alertar o leitor sobre sua existência.

Apesar dos pontos de vista expressos acima, entretanto, as abordagens hierárquica e de rede são obviamente de extrema importância, e têm a vantagem de possuir vários

* Na terceira edição, o material sobre segurança e integridade foi deslocado para um volume suplementar.

anos de experiência acumulada. Por isso as Partes 3 e 4 lidam com essas abordagens em certo detalhe (e espero que, adequadamente, em meu detrimento). A Parte 3 está totalmente baseada em um sistema existente, o IBM Information Management System (IMS), que já se encontra em operação bem-sucedida em várias instalações de computadores. Este sistema foi escolhido como base para a Parte 3 por ser um bom exemplo da abordagem hierárquica e, naturalmente, pela importância intrínseca desse sistema. Por motivos semelhantes, a Parte 4 está baseada nas propostas do "Data Base Task Group" do Comitê CODASYL COBOL. Espero, por isso, que as Partes 3 e 4 sirvam não somente como uma introdução geral às abordagens hierárquica e de rede, mas também, especificamente, como ensino dos sistemas IMS e DBTG. No entanto, o objetivo maior deste livro não é instruir sobre sistemas específicos; ao invés disso, tem como objetivo descrever alguns conceitos gerais, usando sistemas específicos basicamente para fins de ilustração. (Por esta razão muitos outros sistemas também importantes estão pouco mais do que mencionados.) Ainda assim, as descrições do IMS e do DBTG, em particular, apresentam uma quantidade bastante razoável de detalhes.* O leitor que não estiver interessado nos detalhes mais refinados desses sistemas pode omitir certas partes do texto caso deseje, principalmente os Capítulos 19-22, e certas seções (adequadamente indicadas) do Capítulo 24.

Uma nota sobre a terminologia. Como em muitos outros assuntos novos, o campo de sistemas de bancos de dados ainda não possui uma nomenclatura estabelecida. Em particular, a terminologia do IMS difere em muitos pontos da do DBTG. Este livro tenta reconciliar as diferenças relacionando tanto a terminologia do IMS quanto a do DBTG a uma terminologia "neutra" definida nas Partes 1 e 2. (Uma vez isto feito, entretanto, a terminologia "correta" para cada sistema é geralmente empregada nas discussões subsequentes.) A terminologia das Partes 1 e 2, por sua vez, é um amálgama derivado de muitas fontes.

Alguns poucos pontos adicionais sobre a estrutura do livro:

1. Tentei escrever um livro texto, não um trabalho de referência. Naturalmente esses dois objetivos não são totalmente incompatíveis — sem dúvida espero que ambos tenham sido alcançados em grande extensão — mas, onde quer que eles colidam, a tendência foi para o primeiro mais do que para o segundo. Com esta finalidade não hesitei em omitir pontos menores no interesse da clareza, nem em simplificar outros pela mesma razão, muito embora como regra geral eu tenha tentado ser tão completo quanto possível. (O leitor encontrará referências para outros locais para maiores detalhes quando apropriado.)

2. Em sendo um livro-texto, a maioria dos capítulos é seguida por um conjunto de exercícios, que recomendamos ao leitor tentar resolver, pelo menos alguns. As respostas, algumas vezes fornecendo informações adicionais sobre o assunto em questão, encontram-se no final do livro.

3. Cada capítulo é seguido por uma lista de referências, muitas das quais com anotações. As referências estão identificadas no texto por colchetes quadrados. Por exemplo, [1.3] refere-se ao terceiro item na lista de referências no final do capítulo 1, a saber, um artigo do Comitê de Sistemas CODASYL publicado no *BCS Computer Bulletin*, Vol. 15, nº 4, e também no *Communications of the ACM*, Vol. 14, nº 5.

Resta somente a tarefa agradável de reconhecer a ajuda que recebi ao escrever este livro. Sou grato, inicialmente, ao Dr. Codd por uma grande dose de encorajamento, pela

* O mesmo é verdade para o sistema R na terceira edição.

permissão de utilizar boa parte de seu material publicado, particularmente na Parte 2, e por seus comentários úteis no rascunho inicial. As seguintes pessoas também tiveram a gentileza de ler o rascunho e apresentar muitas críticas e sugestões válidas: Joel Aron, Jan Hazelzet, Roger Holliday, Paul Hopewell, Larry Lewis, Salah Mandil, Bill McGee, Herb Meltzer, John Nicholls, Terry Rogers e Tom Work. Gostaria também de agradecer ao Professor Julius T. Tou, o organizador do 4º Simpósio Internacional de Computação e Ciência da Informação (Miami Beach, Florida, 14-16 de dezembro de 1972), e à Plenum Publishing Corporation (editores dos anais) pela permissão para usar um artigo que apresentei naquele simpósio como base para o Capítulo 3. Devo também agradecer aos muitos estudantes IBM cujos comentários no curso original do qual este livro se originou foram muito úteis. Finalmente, sou grato à IBM por permitir que grande parte do preparo do livro fosse feito usando tempo e recursos da companhia. Tenho que enfatizar, entretanto, que sou inteiramente responsável pelo conteúdo do livro; os pontos de vista expressos são meus e de nenhuma forma representam uma posição oficial por parte da IBM.

C. J. D.

Palo Alto, California, Novembro de 1974 (revisado em março de 1981).

Prefácio da Segunda Edição

Ocorreram muitas mudanças no campo do desenvolvimento dos bancos de dados desde que foi escrita a primeira edição. A linguagem de manipulação de dados DBTG e a linguagem de descrição de dados subesquema foram aceitas pelo Comitê CODASYL COBOL para incorporação ao COBOL, e estão comercialmente disponíveis vários sistemas baseados no DBTG. Foram adicionados ao IMS índices secundários e vários outros dispositivos. Começaram a ficar disponíveis sistemas comerciais baseados nos conceitos relacionais. Estão sendo desenvolvidas diversas atividades de padronização. Talvez o fato mais significativo seja o de que universidades e instituições similares através do mundo estão demonstrando um nível de interesse sem precedentes no assunto. A presente edição representa uma tentativa de refletir parte desta atividade. Ela inclui uma grande quantidade de material novo, o que é naturalmente a sua razão de ser; no entanto, a oportunidade está também sendo aproveitada para corrigir alguns erros da primeira edição e melhorar a apresentação em muitas partes. Foram também incluídas muitas referências novas, a maior parte delas com anotação.

Estão sumarizadas abaixo algumas das diferenças mais significativas entre esta edição e a anterior.

Parte 1: Foi revista toda a arquitetura de sistemas para incorporar a terminologia ANSI/SPARC. A apresentação comparativa entre as três abordagens foi unificada e estendida.

Parte 2: O tratamento da estrutura de dados relacional foi expandida para um capítulo à parte; foram adicionados capítulos sobre SEQUEL e Query By Example; e o capítulo sobre normalização adicional foi totalmente reescrito, incluindo agora um tratamento melhor sobre a terceira forma normal e a nova quarta forma normal. Todos os outros capítulos foram consideravelmente revistos.

Parte 3: Foi introduzido um capítulo sobre indexação secundária. Outros capítulos foram revistos para se ajustarem à mais nova versão do IMS.

Parte 4: Todos os capítulos foram revistos para incorporarem mudanças feitas pelos Comitês Data Description Language e COBOL do CODASYL.

*Parte 5**: Os dois capítulos foram revistos para incorporarem mudanças feitas ao DBTG e ao IMS. Foram incluídos sistemas relacionais adicionais. Foi grandemente expandido o tratamento de restrições de integridade e de concorrência.

Parte 6: Esta parte é completamente nova.

Novamente é um grande prazer reconhecer a ajuda que recebi para produzir este livro. Fico particularmente satisfeito por ter a oportunidade de agradecer ao grande número de pessoas que fizeram comentários favoráveis sobre a primeira edição e me encorajaram a expandi-la para a forma presente. Sob este aspecto, eu gostaria especialmente de mencionar Ted Codd, Frank King, Ben Shneiderman e Mike Stonebraker. Estou também profundamente grato às pessoas que se seguem, pela ajuda que me deram em numerosas questões técnicas e pela revisão e crítica de várias partes do rascunho desta edição: David Beech, Don Chamberlin, Rod Cuff, Bob Engles, Ron Fagin, Peter Hitchcock, Roger Holliday, Bill Kent, Bill Lockhart, Ron Obermarck, Vern Watts e Moshe Zloof. Como na primeira edição, estou extremamente agradecido à IBM pelo apoio que me deu neste trabalho. Gostaria também de agradecer à Technical Publishing Company, editores de *Datamation*, pela permissão para basear as revisões no capítulo 3 em um artigo que apareceu naquele jornal em abril de 1976; e à ACM pela permissão para basear partes do capítulo 6 em material em três artigos (referências [26.1], [27.1], e [28.3]), dos quais a ACM tem o *copyright*. Finalmente, gostaria de expressar minha admiração ao *staff* da Addison-Wesley pelo tremendo entusiasmo, encorajamento e paciência demonstrados durante toda a produção das duas edições.

San Jose, California

Junho de 1977 (revisado em março de 1981)

C. J. D.

* Na terceira edição este material foi deslocado para um volume suplementar.

Prefácio da Terceira Edição

O campo da tecnologia de banco de dados continua a evoluir a uma razão sempre crescente; de tal forma que, de fato, os diversos subcampos mais ou menos distintos dessa área começam a emergir como disciplinas próprias. Ao mesmo tempo, há indícios de que algumas das áreas mais antigas estão caminhando para uma situação comparativamente estável. Embora o momento seja claramente adequado para uma nova edição deste livro, tornou-se infelizmente inexequível tratar de todo o assunto da melhor maneira em um único volume, mantendo esse volume com um tamanho razoável. Por isso o material foi dividido em duas partes, que podem ser categorizadas, *grosso modo*, como "básica" e "avançada"; o presente volume contém o material "básico", tendo sido o material "avançado" deslocado para um volume suplementar. No entanto, o presente volume é por si mesmo um livro completo e independente. De fato, sob muitos aspectos, ele fornece uma substituição total à edição prévia; a única exceção é que o material sobre segurança e integridade (Parte 5 na segunda edição), que de certa forma era menos "básico" do que o restante daquela edição, foi deslocado para o volume suplementar. Esta omissão é contrabalançada no presente livro pela introdução de muito material novo, particularmente por uma grande expansão no tratamento da abordagem relacional.

As maiores diferenças entre esta edição e as anteriores estão summarizadas abaixo. É elucidativo mostrar que, no caso das Partes 1, 2, 3, e 5, as modificações são em sua maior parte esclarecimentos ou adições — embora adições bastante significativas — ao material encontrado no livro anterior. Em contraste, as mudanças na Parte 4 incluem numerosas modificações de *fato* (refletindo modificações nos documentos fonte correspondentes). Em outras palavras, a Parte 4 da segunda edição deve agora ser considerada como obsoleta; as Partes 1, 2, 3, e 5 ainda estão razoavelmente precisas, mas não podem ser vistas como estando totalmente atualizadas.

Parte 1. O capítulo 1 foi totalmente revisto para fornecer uma introdução geral mais suave ao assunto e uma melhor descrição da arquitetura ANSI/SPARC. O capítulo 2 foi estendido para incluir uma discussão sobre as árvores-B e uma breve introdução ao endereçamento randômico extensível.

Parte 2. Esta foi praticamente toda reescrita. O capítulo 4 foi expandido para dar uma explicação melhor sobre a estrutura relacional, e em particular para incorporar duas regras fundamentais de integridade. Os capítulos 5–10 (que substituem os capítulos 7 e 10 da segunda edição) fornecem uma descrição completa do sistema relacional System R, usando-o como veículo para ilustrar numerosos conceitos de sistemas relacionais.* O capítulo 11 sobre Query By Example (Capítulo 8 na segunda edição) foi revisto para ficar mais em linha com o produto QBE disponível da IBM. O capítulo 12 (capítulo 6 na segunda edição) fornece um tratamento bastante mais completo sobre álgebra relacional do que antes. O capítulo 13 (capítulo 5 na segunda edição) apresenta o cálculo relacional, tanto na versão de tupla como na de domínio, de uma forma algo mais formal do que antes, e usa QUEL (a linguagem do sistema INGRES) para seus exemplos ao invés de DSL ALPHA, que nunca foi implementada. O capítulo 14 (capítulo 9 na segunda edição) foi estendido para incluir o conceito de decomposições boas ou más, a quinta forma normal “final”, e um tratamento muito melhorado da quarta forma normal. Finalmente, o capítulo 11 na segunda edição (“Alguns Sistemas Relacionais”) foi removido, embora a maior parte do seu conteúdo ainda sobreviva na forma de anotações na bibliografia.

Parte 3. Esta foi estendida para incluir material sobre sensitividade a nível de campo (Capítulo 17), uso de múltiplos PCBs (Capítulo 18), e um novo capítulo sobre bancos de dados Fast Path (Caminho Rápido) (Capítulo 22).

Parte 4. Este material foi totalmente revisto em concordância com as especificações mais recentes do CODASYL e documentos de trabalho do Comitê COBOL (X3J4) e do Comitê ANSI Data Description Language (X3H2). Em adição, foram significativamente aperfeiçoadas as respostas aos exercícios.

Parte 5. (Parte 6 na segunda edição). O capítulo 27, um tratamento completo da Unified Database Language UDL, é uma versão grandemente expandida da antiga Seção 25.2. O capítulo 28 é uma versão estendida da antiga Seção 25.3.

Uma alteração adicional é que o termo “modelo de dados”, muito usado nas edições anteriores, foi abandonado em larga escala no presente livro. Este termo infelizmente recebeu uma diversificação de significados por elementos militantes na área e por isso tornou-se fonte de algumas confusões. Nem eu posso me considerar inocente sob este aspecto: O termo aparece (no Capítulo 12) com um significado um pouco diferente do que lhe foi atribuído nas edições anteriores. A terminologia de sistemas de bancos de dados ainda está, infelizmente, em estado de curso.

Agradeço aos muitos amigos e colegas que me ajudaram oferecendo sugestões ou revendo e comentando partes do rascunho desta edição – em particular a David Beech, Don Chamberlin, Ted Codd, Bob Engles, Ron Fagin, Bill Kent, Pete Lazarus, Jim Pantaja, Franz Remmen, Reind van de Riet, Bob Smead, e, muito especialmente, Phil Shaw, que executou o trabalho mais completo de revisão do manuscrito que qualquer autor poderia desejar. Gostaria também de agradecer a Karen Takle Quinn e Per Groth pelo auxílio que me deram no uso da bibliografia da IBM Santa Teresa sobre o sistema QBE, e Paul Pittman, Russ Williams e membros do grupo de desenvolvimento do Sistema R por me ajudarem em experiências com o Sistema R. Como nas edições anteriores, é com prazer

* Quando este livro estava indo para o prelo (janeiro de 1981) um novo programa-produto para banco de dados conhecido como SQL/DS foi anunciado pela IBM. O SQL/DS, que funciona sob o sistema operacional DOS/VSE, incorpora praticamente todos os dispositivos do sistema R.

que reconheço o apoio que recebi da IBM para escrever este livro. Quero também agradecer à ACM pela permissão de incluir algumas partes de três artigos (referências [1.14], [4.6.], e [5.1.] dos quais a ACM detém o *copyright*. E, finalmente, é uma real satisfação novamente expressar minha gratidão pela amizade, cooperação e profissionalismo mostrados por todos na Addison-Wesley durante toda a produção desta edição.

*San Jose, California
Março de 1981*

C. J. D.

Sumário

PARTE 1 ARQUITETURA DE SISTEMAS DE BANCOS DE DADOS

Capítulo 1

Conceitos Básicos

1.1	O que é um Sistema de Banco de Dados?	26
1.2	Dados Operacionais	29
1.3	Por que Banco de Dados?	31
1.4	Independência de Dados	34
1.5	Uma Arquitetura para Sistemas de Banco de Dados	38
1.6	Bancos de Dados Distribuídos	47
	Exercícios	48
	Referências e Bibliografia	48

Capítulo 2

Estruturas de Armazenamento

2.1	Introdução	51
2.2	Representações Possíveis para Alguns Dados de Teste	54
2.3	Interface do Registro Físico: Técnicas de Indexação.	61
2.4	Técnicas Gerais de Indexação.	66
	Exercícios	70
	Referências e Bibliografia	71

Capítulo 3

Estruturas de Dados e Operadores Correspondentes

3.1	Introdução.	76
3.2	A Abordagem Relacional	77
3.3	A Abordagem Hierárquica.	79
3.4	A Abordagem de Rede	82
3.5	Operadores de Nível mais Alto.	85

3.6	Resumo	89
	Exercícios	90
	Referências e Bibliografia	91
PARTE 2		
A ABORDAGEM RELACIONAL		
Capítulo 4		
Estrutura Relacional de Dados		
4.1	Relações	93
4.2	Domínios e Atributos.	95
4.3	Chaves.	97
4.4	Extensões e Intensões.	99
4.5	Resumo	100
	Exercícios	102
	Referências e Bibliografia	102
Capítulo 5		
A Arquitetura do Sistema R		
5.1	Histórico	104
5.2	Arquitetura	105
	Referências e Bibliografia	110
Capítulo 6		
Estrutura de Dados do Sistema R		
6.1	Introdução.	112
6.2	Tabelas Básicas	112
6.3	Índices	114
6.4	Discussão.	115
	Exercícios	117
	Referências e Bibliografia	118
Capítulo 7		
Manipulação de Dados do Sistema R		
7.1	Introdução.	119
7.2	Operações de Recuperação	119
7.3	Funções Integradas	132
7.4	Operações de Atualização.	135
7.5	Dicionário do Sistema R.	137
7.6	Discussão.	139
	Exercícios	140
	Referências e Bibliografia	142
Capítulo 8		
SQL Embutida		
8.1	Introdução.	143

8.2	Operações não Envolvendo Cursos	145
8.3	Operações Envolvendo Cursos	146
8.4	Instruções Dinâmicas	150
8.5	Discussão	153
	Exercícios	154
	Referências e Bibliografia	154

Capítulo 9

O Nível Externo do Sistema R

9.1	Introdução	155
9.2	Visões	155
9.3	Operações DML Sobre Visões	157
9.4	Visões e Independência de Dados	161
9.5	Resumo	163
	Exercícios	163
	Referências e Bibliografia	163

Capítulo 10

O Nível Interno do Sistema R

10.1	O Research Storage System	165
10.2	Segmentos e Páginas	165
10.3	Arquivos e Registros	166
10.4	Caminhos de Acesso	168
10.5	Um Exemplo	170
10.6	O Diretório RSS	172
	Referências e Bibliografia	172

Capítulo 11

Query By Example

11.1	Introdução	173
11.2	Operações de Recuperação	174
11.3	Operações de Recuperação em Relações de Estrutura em Árvore	178
11.4	Funções Integradas	182
11.5	Operações de Atualização	184
11.6	O Dicionário QBE	186
11.7	Discussão	189
	Exercícios	191
	Referências e Bibliografia	191

Capítulo 12

Álgebra Relacional

12.1	Introdução	193
12.2	Operações Tradicionais com Conjuntos	195
12.3	Nomes-Atributo para Relações Derivadas	196
12.4	Operações Relacionais Especiais	197
12.5	Exemplos	201

12.6	Discussão	202
	Exercícios	203
	Referências e Bibliografia	203
 Capítulo 13		
Cálculo Relacional		
13.1	Introdução	209
13.2	Cálculo Relacional Orientado para Tupla	210
13.3	Cálculo Relacional Orientado para Domínio	216
	Exercícios	218
	Referências e Bibliografia	219
 Capítulo 14		
Normalização Adicional		
14.1	Introdução	222
14.2	Dependência Funcional	224
14.3	Primeira, Segunda e Terceira Formas Normais	227
14.4	Relações com mais de uma Chave Candidata	233
14.5	Decomposições Boas e MÁS	236
14.6	Quarta Forma Normal	238
14.7	Quinta Forma Normal	242
14.8	Resumo	245
	Exercícios	247
	Referências e Bibliografia	249
 PARTE 3		
A ABORDAGEM HIERÁRQUICA		
 Capítulo 15		
A Arquitetura de um Sistema IMS		
15.1	Histórico	254
15.2	Arquitetura	254
	Referências e Bibliografia	256
 Capítulo 16		
Estrutura de Dados IMS		
16.1	Bancos de Dados Físicos	257
16.2	A Descrição do Banco de Dados	260
16.3	Seqüência Hierárquica	262
16.4	Algumas Observações sobre o Banco de Dados de Educação	262
	Exercícios	264
	Referências e Bibliografia	264
 Capítulo 17		
O Nível Externo do IMS		
17.1	Bancos de Dados Lógicos	265

17.2	O Program Communication Block	267
	Exercício	269
	Referências e Bibliografia	269

Capítulo 18

Manipulação de Dados IMS

18.1	Definindo o Program Communication Block (PCB)	270
18.2	Operações DL/I	272
18.3	Exemplos DL/I	273
18.4	Construindo o Segment Search Argument (SSA)	278
18.5	Códigos de Comando SSA	278
18.6	Usando mais de um PCB	281
	Exercícios	282
	Referências e Bibliografia	283

Capítulo 19

O Nível Interno do IMS

19.1	Introdução	284
19.2	HSAM	286
19.3	HISAM	287
19.4	Estruturas HD: Indicadores de Localização	291
19.5	HDAM	293
19.6	HIDAM	296
19.7	Grupos de Arquivos Secundários	297
19.8	A Definição de Mapeamento	301
19.9	Reorganização	304
19.10	Independência de Dados	305
19.11	Resumo	306
	Exercícios	307
	Referências e Bibliografia	308

Capítulo 20

Bancos de Dados Lógicos do IMS

20.1	Logical Databases (LDBs)	309
20.2	Um Exemplo	310
20.3	Terminologia	313
20.4	Os Database Descriptions (DBDs)	314
20.5	Carga do Banco de Dados Lógico	316
20.6	Processamento do Banco de Dados Lógico	317
20.7	Relacionamentos Lógicos Bidirecionais	319
20.8	Uma Observação sobre a Estrutura de Armazenamento	324
20.9	Bancos de Dados Lógicos Envolvendo um Único Banco de Dados Físico	324
20.10	Algumas Regras e Restrições	327
20.11	Resumo	328
	Exercícios	329
	Referências e Bibliografia	329

Capítulo 21 **Índices Secundários do IMS**

21.1	Introdução	330
21.2	Indexando a Raiz em um Campo que não o Campo de Seqüência	331
21.3	Indexando a Raiz em um Campo em um Dependente	335
21.4	Indexando um Dependente em um Campo Daquele Dependente	337
21.5	Indexando um Dependente em um Campo em um Dependente de Nível mais Baixo	339
21.6	Dispositivos Adicionais	339
21.7	Resumo	340
	Exercícios	341
	Referências e Bibliografia	341

Capítulo 22 **Bancos de Dados IMS Fast Path**

22.1	O Dispositivo Fast Path	342
22.2	Bancos de Dados na Memória Principal	343
22.3	Bancos de Dados Data Entry	348
	Referências e Bibliografia	350

PARTE 4 **A ABORDAGEM DE REDE**

Capítulo 23 **A Arquitetura de um Sistema DBTG**

23.1	Histórico	352
23.2	Arquitetura	353
	Referências e Bibliografia	354

Capítulo 24 **Estrutura de Dados DBTG**

24.1	Introdução	355
24.2	Formação do Conjunto: Exemplos Hierárquicos	356
24.3	Formação do Conjunto: Exemplos em Rede	362
24.4	Conjuntos Singulares	366
24.5	Um Exemplo de Esquema	367
24.6	Classe de Membro	371
24.7	Seleção de Conjunto	374
	Exercícios	376
	Referências e Bibliografia	376

Capítulo 25 **O Nível Externo do DBTG**

25.1	Introdução	379
25.2	Diferenças Entre Subesquema e Esquema	379

25.3	Um Exemplo de Subesquema	380
	Exercício	381
	Referências e Bibliografia	382

Capítulo 26

Manipulação de Dados DBTG

26.1	Introdução	383
26.2	Indicação Corrente	383
26.3	Manuseio de Exceções	386
26.4	GET	387
26.5	STORE	387
26.6	ERASE	388
26.7	MODIFY	389
26.8	CONNECT	390
26.9	DISCONNECT	390
26.10	RECONNECT	391
26.11	FIND	391
26.12	Listas de Retenção	397
26.13	Instruções Diversas	400
	Exercícios	401
	Referências e Bibliografia	402

PARTE 5

REVISÃO DAS TRÊS ABORDAGENS

Capítulo 27

A Linguagem Unificada de Banco de Dados

27.1	Introdução	404
27.2	A Abordagem da Conjunção	404
27.3	Linguagem Declarativa	407
27.4	Linguagem Manipulativa	411
27.5	Dispositivos Adicionais	418
27.6	Conclusão	421
	Exercícios	422
	Referências e Bibliografia	423

Capítulo 28

Uma Comparação entre as Abordagens Relacional e de Rede

28.1	Introdução	425
28.2	O Nível Conceitual	425
28.3	Critério para o Esquema Conceitual	427
28.4	A Abordagem Relacional	429
28.5	A Abordagem em Rede	430
28.6	Conclusão	437
	Referências e Bibliografia	437

Respostas a Exercícios Selecionados	440
Lista de Abreviaturas	501
Índice Analítico	503

Parte 1

Arquitetura de Sistemas

de Bancos de Dados

A parte 1 consiste de três capítulos introdutórios. O capítulo 1 estabelece o panorama explicando o que é um banco de dados e definindo um esboço de arquitetura para um sistema de banco de dados. Esta arquitetura servirá como uma estrutura básica sobre a qual serão montados capítulos posteriores deste livro. O capítulo 2 é uma breve introdução a algumas técnicas voltadas para o arranjo físico de dados armazenados no banco de dados. O capítulo 3 é provavelmente o mais importante; ele trata de um problema que é fundamental no projeto de qualquer sistema de banco de dados, que é o de como esse banco de dados deve ser visto pelos usuários. (É normal que os usuários fiquem isolados de detalhes referentes a como o dado está fisicamente armazenado para permitir que eles vejam o banco de dados de uma forma mais adequada às suas necessidades.) O capítulo 3 faz a introdução às três maiores abordagens a este problema, que são a relacional, a hierárquica e a de rede, preparando o caminho para as três partes seguintes do livro.

1

Conceitos Básicos

1.1 O QUE É UM SISTEMA DE BANCO DE DADOS?

A tecnologia de banco de dados já foi descrita como sendo “uma das áreas de mais rápido crescimento na ciência da computação e da informação” [1.14]. Como um campo, ela ainda é comparativamente jovem; os fabricantes e vendedores só começaram a oferecer sistemas de gerenciamento de bancos de dados no final da década de 1960 (embora seja verdade que alguns pacotes de *software* anteriores não incluíssem diversas funções hoje associadas com tais sistemas [1.13, 1.15]). Entretanto, apesar da sua recenteidade, o campo tornou-se rapidamente de considerável importância, tanto no plano prático como no teórico. A quantidade total de dados hoje comprometida com bancos de dados pode, de forma conservadora, ser medida em bilhões de *bytes*; os investimentos financeiros envolvidos representam um valor correspondentemente grande; e não há exagero em dizer-se que milhares de organizações tornaram-se criticamente dependentes da operação contínua e bem-sucedida do sistema de banco de dados.

Então, o que é exatamente um sistema de banco de dados? Basicamente nada mais é do que um sistema de armazenamento de dados baseado em computador; isto é, um sistema cujo objetivo global é registrar e manter informação.¹ Esta informação pode ser qualquer uma considerada significativa à organização servida pelo sistema — em outras palavras, qualquer uma necessária ao processo de decisão da gerência daquela organização. A figura 1.1 mostra uma visão bastante simplificada de um sistema de banco de dados.

¹ Os termos “*dado*” e “*informação*” são tratados como sinônimos neste livro. Alguns autores fazem uma distinção entre os dois, usando o termo “*dado*” para se referir aos valores fisicamente registrados no banco de dados, e “*informação*” para se referir ao *significado* desses valores para algum usuário. A distinção é claramente importante — tão importante que parece preferível torná-la explícita, onde isto for relevante, ao invés de confiar em uma diferenciação algo arbitrária entre os dois termos essencialmente similares.

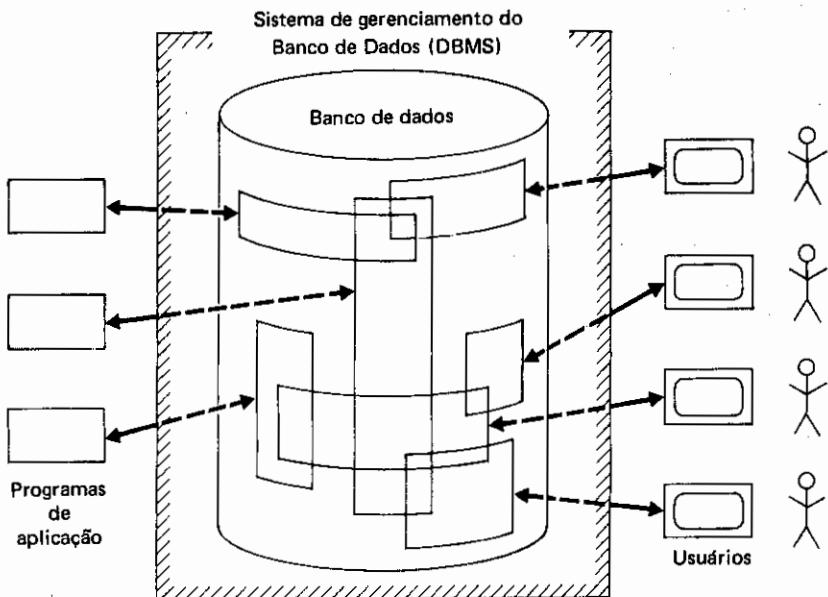


Fig. 1.1 Imagem simplificada de um sistema de banco de dados

A figura 1.1 pretende mostrar que um sistema de banco de dados envolve quatro componentes maiores: dados, hardware, software, e usuários. Vamos tecer considerações breves sobre cada um mais adiante. Posteriormente neste capítulo vamos discutir cada um com maior riqueza de detalhes.

Dados

Os dados armazenados no sistema são repartidos em um ou mais *bancos de dados*. Para fins didáticos é conveniente assumir-se que há somente um banco de dados, contendo a totalidade dos dados armazenados no sistema, e nós normalmente usaremos esta hipótese simplificadora, pois ela não invalida substancialmente qualquer das discussões subsequentes. Há boas razões para que esta restrição não seja imposta na prática, entretanto, como veremos mais tarde.

Portanto, um banco de dados é um depósito de dados armazenados. Geralmente ele é tanto integrado como compartilhado.

Por “integrado” queremos dizer que o banco de dados pode ser imaginado como sendo a unificação de diversos arquivos que, de outra forma, seriam distintos, eliminando parcial ou totalmente qualquer redundância entre aqueles arquivos. Por exemplo, um determinado arquivo poderia conter tanto registros de EMPREGADOS, dando nome, endereço, departamento, salário, etc., como registros de MATRÍCULA, representando a matrícula de empregados em cursos de treinamento. Imaginemos que para o processo de administração de cursos seja necessário conhecer-se o departamento de cada estudante matriculado. Claramente não é preciso incluir esta informação, redundantemente, nos registros de

BD
INTEG

utlizamos multivisualizações

MATRÍCULAS, pois ela pode ser sempre encontrada por pesquisa nos registros correspondentes de EMPREGADOS.

Por "compartilhado" queremos dizer que partes individuais de dados podem ser compartilhadas entre diversos usuários diferentes, significando que cada um daqueles usuários pode ter acesso à mesma parte do dado (e pode usá-la para finalidades diferentes). O compartilhamento é na realidade uma consequência do fato de ser o banco de dados integrado; no exemplo EMPREGADO/MATRICULA citado acima, a informação sobre departamento nos registros de EMPREGADO é compartilhada pelos usuários do departamento de pessoal e pelos usuários do departamento de educação. Outra consequência do mesmo fato (ser o banco de dados integrado) é a de que qualquer usuário específico estará normalmente interessado somente em um subconjunto do banco de dados total; além disso, os subconjuntos de diferentes usuários irão se superpor de muitas maneiras diferentes. Em outras palavras, um determinado banco de dados será percebido por usuários diferentes sob uma variedade de formas diferentes. (Mesmo quando dois usuários compartilham o mesmo subconjunto do banco de dados, suas visões daquele subconjunto podem diferir consideravelmente a nível de detalhe. Este tópico será mais amplamente discutido na Seção 1.4.)

O termo "compartilhado" é freqüentemente expandido para cobrir não somente o compartilhamento como foi descrito, mas também o compartilhamento concorrente; isto é, a capacidade de que diversos usuários diferentes estejam tendo acesso conjunto ao banco de dados — possivelmente à mesma parte do dado — *ao mesmo tempo*. (Um sistema de banco de dados que suporte esta forma de compartilhamento é algumas vezes conhecido como um sistema de multiplos usuários.)

Hardware

O hardware consiste dos volumes de memória secundária — discos, tambores, etc. — nos quais resida o banco de dados, juntamente com os dispositivos associados, unidades de controle, canais e assim por diante. (Estamos partindo do princípio de que o banco de dados é muito grande para caber totalmente na memória principal do computador.) Este livro não se detém muito nos aspectos de hardware do sistema, pelas seguintes razões: primeiramente, esses aspectos representam por si mesmos um tópico maior; segundo que os problemas encontrados nessa área não são peculiares aos sistemas de bancos de dados; e terceiro, esses problemas já foram investigados extensamente e estão documentados em outras partes.

Software — SGBD

Entre o banco de dados físico (isto é, os dados armazenados) e os usuários do sistema encontra-se uma camada de software, usualmente chamada de sistema de gerenciamento do banco de dados ou DBMS. Todas as solicitações dos usuários para acesso ao banco de dados são manipuladas pelo DBMS. Outra função geral provida pelo DBMS é portanto isolar os usuários do banco de dados dos níveis de detalhes de hardware (de forma semelhante àquela em que as linguagens de programação tais como o COBOL isolam os usuários programadores dos detalhes a nível de hardware). Em outras palavras, o DBMS fornece uma visão do banco de dados elevada algo acima do nível de hardware, e suporta a operação do usuário (tal como "leia o registro EMPREGADO do empregado Smith") expressa em termos daquela visão em nível mais alto. Nós iremos discutir esta e outras funções do DBMS mais tarde com muito mais detalhe.

Usuários

Nós consideramos três grandes classes de usuários. Primeiramente temos o programador de aplicações, responsável por escrever os programas de aplicação que utilizam o banco de dados, tipicamente em linguagens como o COBOL ou o PL/I. Estes programadores de aplicação operam com os dados de todas as formas usuais: recuperando informação, criando nova informação, retirando ou alterando informação existente. (Todas essas funções são executadas pela emissão de solicitações adequadas ao DBMS.) Os programas em si podem ser de aplicações convencionais em lotes (*batch*) ou podem ser programas “*on-line*”, que são projetados para suportar o usuário (veja abaixo) interagindo com o sistema a partir de um terminal.

A segunda classe de usuário, então, é o usuário final, que tem acesso ao banco de dados a partir de um terminal. Um usuário final pode utilizar uma *linguagem de consulta* fornecida como parte integrante do sistema, ou (como mencionado acima) pode chamar uma aplicação escrita pelo usuário sob a forma de um programa que aceita comandos de um terminal e por sua vez emite solicitações ao DBMS de acordo com o comando pelo usuário final. De qualquer forma, o usuário novamente pode, em geral, executar todas as funções de recuperação, criação, eliminação, ou modificação, embora seja provavelmente verdadeiro dizer que recuperação é a função mais comum desta classe de usuário.

A terceira classe de usuário é o administrador do banco de dados, ou DBA (não está mostrado na figura 1.1). A discussão sobre a função do DBA fica transferida para a Seção 1.5.

Assim completamos nossa descrição preliminar sobre os aspectos maiores de um sistema de banco de dados. O restante deste capítulo irá aprofundar alguns detalhes sobre esses tópicos.

1.2 DADOS OPERACIONAIS

(Engles, 1973)

Em um dos primeiros estudos sobre o assunto [1.16], Engles refere-se aos dados no banco de dados como “dados operacionais”, distinguindo-os de dados de entrada, dados de saída e outros tipos de dados. Abaixo damos uma versão modificada da definição original de Engles sobre *banco de dados*:

- Um banco de dados é uma coleção de dados operacionais armazenados usados pelo sistema de aplicações de uma empresa específica.

Esta definição requer alguma explicação. “Empresa” é simplesmente um termo genérico conveniente para designar uma organização comercial, científica, técnica ou de outra natureza que seja razoavelmente auto-suficiente. Alguns exemplos são:

Companhias de manufatura,
Bancos,
Hospitais,
Universidades,
Departamentos do governo.

Qualquer empresa tem necessariamente que manter uma quantidade de dados sobre suas operações. Estes são os “dados operacionais”. Os dados operacionais para as empresas listadas acima provavelmente incluiriam o seguinte:

Dados sobre produtos,
Dados sobre contabilidade,
Dados sobre pacientes,

Dados sobre estudantes,
Dados de planejamento.

Como já mencionado, os dados operacionais *não* incluem dados de entrada ou saída, filas de trabalho, ou, sem dúvida, qualquer informação puramente transiente. "Dados de entrada" são as informações que entram no sistema a partir do mundo exterior (tipicamente em cartões ou a partir de um terminal); essas informações podem provocar a ocorrência de modificações dos dados operacionais, mas por si mesmas não fazem parte do banco de dados. De maneira semelhante, "dados de saída" são mensagens e relatórios que emanam do sistema (impressos ou projetados em um terminal); novamente, esses relatórios contêm informações derivadas dos dados operacionais, mas por si mesmos não fazem parte do banco de dados.

Como uma ilustração do conceito de dados operacionais, vamos considerar o caso de uma companhia de manufatura com um pouco mais de detalhes. Esta empresa desejará manter informações sobre os *projetos* que possui; as *peças* usadas nesses projetos; os *fornecedores* dessas peças; os *depósitos* onde essas peças estão estocadas; os *empregados* que executam os projetos; e assim por diante. Estas são as entidades básicas sobre as quais são registrados dados no banco de dados. (O termo "entidade" é amplamente usado em sistemas de bancos de dados para significar qualquer objeto distingível que deva ser representado no banco de dados.) Veja Fig. 1.2.

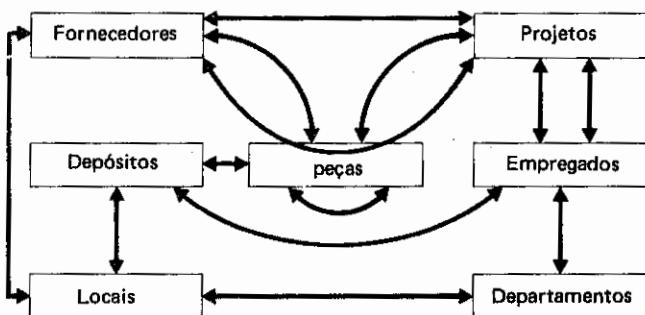


Fig. 1.2 Um exemplo de dados operacionais

É importante notar que geralmente haverá *associações* ou *relacionamentos* interligando as entidades básicas entre si. Elas são representadas por setas de conexão na Fig. 1.2. Por exemplo, há uma associação entre fornecedores e peças: cada fornecedor fornece certas peças e, por seu turno, cada peça é suprida por certos fornecedores. Semelhantemente, peças são usadas em projetos e, por seu turno, projetos usam peças; peças são armazenadas em depósitos, e depósitos armazenam peças; e assim por diante. Note que todos estes relacionamentos são bidirecionais; isto é, eles podem ocorrer nas duas direções. Por exemplo, o relacionamento entre empregados e departamentos pode ser usado para responder às duas seguintes perguntas: a) Dado um empregado, encontre o departamento correspondente; b) Dado um departamento, encontre todos os empregados correspondentes.

O ponto significativo sobre relacionamentos como os ilustrados na Fig. 1.2 é que eles *são parte dos dados operacionais tanto quanto as entidades associadas*. Conseqüentemente eles têm que estar representados no banco de dados. Mais tarde iremos considerar as várias maneiras de como isto pode ser feito.

A figura 1.2 ilustra alguns outros pontos.

1. Muito embora a maioria dos relacionamentos no diagrama associe *dois* tipos de entidades, este não é definitivamente sempre o caso. No exemplo há uma seta conectando três tipos de entidades (fornecedores – peças – projetos). Isto poderia representar o fato de que certos fornecedores fornecem peças para certos projetos. Isto *não* é o mesmo que a combinação da associação fornecedores-peças com a associação peças-projetos (em geral). Por exemplo, a informação de que “o fornecedor S2 fornece a peça P4 para o projeto J3” nos diz *mais* do que a combinação “o fornecedor S2 fornece a peça P4” e “a peça P4 é usada no projeto J3” – nós não podemos deduzir a primeira destas três associações conhecendo somente a segunda e a terceira (mas *podemos* deduzir a segunda e a terceira conhecendo a primeira). Mais explicitamente, se soubermos que S2 fornece P4 e que P4 é usada em J3, então podemos deduzir que S2 fornece P4 para algum projeto Jx, e que algum fornecedor Sy fornece P4 para J3, mas não podemos inferir que Jx seja J3 nem que Sy seja S2. Inferências falsas como essas são exemplos do que se pode chamar de *armadilha de conexão*.

2. O exemplo também mostra uma seta envolvendo somente *um* tipo de entidade (peças). Isto representa uma associação entre uma peça e outra; por exemplo, o fato de que algumas peças são componentes de outras peças (um parafuso é um componente de uma dobradiça, que por sua vez é também considerada uma peça).

3. Em geral, as mesmas entidades podem estar associadas em qualquer número de relacionamentos. No exemplo, projetos e empregados estão ligados em dois relacionamentos. Um pode representar o relacionamento “trabalha em” (o empregado trabalha no projeto), e o outro relacionamento “é o gerente de” (um empregado é o gerente do projeto).

Muitos textos (e sistemas) de bancos de dados consideram entidades e relacionamentos como dois tipos fundamentalmente heterogêneos de objetos. Entretanto, uma associação entre entidades pode por si mesma ser considerada como uma entidade. Se tomarmos como definição de entidade “um objeto sobre o qual se deseja registrar informação”, então uma associação certamente se encaixa na definição. Por exemplo, “a peça P4 está armazenada no depósito W8” é uma entidade sobre a qual desejamos registrar informação, isto é, a quantidade existente. Por isso neste livro vamos tender a ver relacionamentos como simples tipos especiais de entidade.

1.3 POR QUE BANCO DE DADOS?

Por que deveria uma empresa optar por armazenar seus dados operacionais em um banco de dados integrado? Uma resposta, *grosso modo*, a esta questão (elaborada abaixo) é a de que um sistema de banco de dados proporciona à empresa um *controle centralizado* de seus dados operacionais – que como o leitor deve perceber é um dos seus ativos mais valiosos. Isto contrasta vivamente com a situação que prevalece hoje em muitas empresas, onde tipicamente cada aplicação dispõe de seus arquivos privativos – muito freqüentemente também fitas e discos privativos – de tal forma que os dados operacionais estão ‘garmente dispersos, tornando provavelmente difícil o controle.

O antecedente implica que em uma empresa com um sistema de banco de dados exista alguém identificável que tenha esta responsabilidade central pelos dados operacionais. Esta pessoa é o administrador do banco de dados (DBA) mencionado na Seção 1.1. Nós vamos discutir o papel do DBA e detalhes mais tarde; por enquanto é suficiente notarmos que seu trabalho requer tanto um alto grau de capacitação técnica quanto de entender e interpretar necessidades de gerências de alto nível. (Na prática, DBA pode consistir de um time de pessoas ao invés de apenas uma.) É importante perceber que a posição do DBA dentro de uma empresa está em nível bastante alto.

Vamos considerar algumas vantagens que advêm da existência do controle centralizado dos dados, conforme discutido acima.

2 • A redundância pode ser reduzida.

Em sistemas sem bancos de dados, cada aplicação possui seus arquivos privativos. Isto freqüentemente gera uma redundância considerável nos dados armazenados, com resultante perda de espaço de armazenamento. Por exemplo, uma aplicação de pessoal e uma aplicação de registros de educação podem cada uma ter seu arquivo contendo informação de departamento dos empregados. Como nós sugerimos na Seção 1.1, esses dois arquivos podem ser integrados e a redundância eliminada se o DBA estiver a par das necessidades de dados para as duas aplicações – isto é, se o DBA possuir o necessário controle geral.

Não pretendemos sugerir que toda redundância deva necessariamente ser eliminada. Algumas vezes existem sólidas razões técnicas ou de negócios para que se mantenham múltiplas cópias do mesmo dado. Entretanto, em sistemas de bancos de dados, a redundância deve ser controlada – isto é, o sistema deve ter conhecimento dessa redundância e assumir a responsabilidade de propagar as atualizações (veja o próximo tópico abaixo).

2 • A inconsistência pode ser evitada (até certo ponto).

Este é realmente um corolário do tópico anterior. Imaginemos que um determinado fato sobre o mundo real – digamos, o fato de que o empregado E3 trabalha no departamento D8 – está representado em duas entradas distintas no banco de dados, e que o sistema não tem conhecimento desta duplicação (em outras palavras, a redundância não é controlada). Então haverá ocasiões em que as duas entradas não serão concordantes (isto é, quando uma e somente uma tiver sido atualizada.) Nesse momento o banco de dados é dito estar inconsistente. Obviamente, um banco de dados em estado de inconsistência é capaz de fornecer informação incorreta ou conflitante.

É claro que se o fato mencionado estiver representado por uma única entrada (isto é, se a redundância for removida), tal inconsistência não pode ocorrer. Alternativamente, se a redundância não for removida mas for controlada (tornando-se conhecida para o sistema), então o sistema pode garantir que o banco de dados nunca estará inconsistente quando visto pelo usuário, ao garantir que qualquer alteração feita em uma das duas entradas seja automaticamente efetuada na outra. Este processo é conhecido como propagação de atualização – onde o termo “atualização” é usado para cobrir todas as operações de criação, remoção ou modificação. (Note, entretanto, que poucos sistemas hoje são capazes de propagar atualizações automaticamente; isto é, a maioria dos sistemas correntes não suporta redundância controlada.)

3 • Os dados podem ser compartilhados.

Nós discutimos este ponto na Seção 1.1, mas ele é tão importante que vamos reforçá-lo novamente aqui. Isto significa não somente que as aplicações existentes podem

compartilhar os dados do banco de dados, mas também que novas aplicações podem ser desenvolvidas para operar sobre os mesmos dados armazenados. Em outras palavras, as necessidades de dados das novas aplicações podem ser satisfeitas sem que se tenha que criar quaisquer novos arquivos armazenados.

4 • Os padrões podem ser reforçados.

Com controle central do banco de dados, o DBA pode garantir que todos os padrões aplicáveis foram seguidos na representação do dado. Os padrões aplicáveis são qualquer um ou todos da companhia, da indústria, da instalação, do departamento, nacionais ou internacionais. É particularmente desejável uma padronização dos formatos dos dados armazenados para facilitar o *intercâmbio de dados* ou migração entre sistemas.

5 • Podem ser aplicadas restrições de segurança.

Possuindo uma completa jurisdição sobre os dados operacionais, o DBA pode: a) Garantir que as únicas vias de acesso ao banco de dados sejam através dos canais adequados, e portanto, b) Pode definir a execução de verificações de autorização sempre que for tentado o acesso a dados sensíveis. Podem ser estabelecidas diferentes verificações para cada tipo de acesso (recuperação, modificação, remoção, etc.) para cada parte da informação do banco de dados. [Note que, sem tais verificações, a segurança dos dados pode na realidade estar sob um risco maior em um sistema de banco de dados do que estaria em um sistema tradicional (disperso) de arquivamento.]

6 • A integridade pode ser mantida.

O problema de integridade é o de se garantir que os dados no banco de dados sejam precisos. A inconsistência entre duas entradas representando o mesmo "fato" é um exemplo de perda de integridade (que naturalmente só pode ocorrer se existir redundância nos dados armazenados.) Entretanto, mesmo com a eliminação da redundância, o banco de dados ainda pode conter dados incorretos. Por exemplo, pode estar registrado que um empregado trabalhou 200 horas na semana, ou uma lista de números de empregados de um dado departamento pode conter o de um empregado inexistente. O controle centralizado do banco de dados ajuda a evitar essas situações, até quanto elas podem ser evitadas, por possibilitar que o DBA defina procedimentos de validação a serem executados sempre que seja tentada uma operação de atualização. (Novamente estamos utilizando o termo "atualização" para cobrir todas as operações de modificação, criação e remoção.) É importante chamar a atenção para o fato de que a integridade de dados é ainda mais importante em sistemas de bancos de dados do que em um ambiente de "arquivos privativos", precisamente porque o banco de dados é compartilhado; pois sem procedimentos apropriados de validação, é possível que um programa contendo erros gere dados ruins e "infecte" outros programas corretos que usam aqueles dados.

7 • Necessidades conflitantes podem ser balanceadas.

Conhecendo as necessidades globais da empresa – ao contrário das necessidades de um usuário individual –, o DBA pode estruturar o sistema de banco de dados para fornecer um serviço global que é "o melhor para a empresa". Por exemplo, pode ser escolhida uma representação para os dados na memória que dê rápido acesso às aplicações mais importantes ao custo de um desempenho pior para algumas outras aplicações.

A maioria das vantagens listadas acima é bastante óbvia. Entretanto, um outro ponto, que não é tão óbvio – muito embora tenha relação com diversos dos anteriores –, tem que ser adicionado à lista; é a *provisão de independência de dados*. (Estritamente fa-

lando, isto é mais um *objetivo* do que uma vantagem.) Este conceito é tão importante que vamos dedicar uma seção separada para ele.

1.4 INDEPENDÊNCIA DE DADOS

A independência de dados pode ser mais facilmente compreendida se considerarmos primeiramente o seu oposto. A maioria das aplicações existentes hoje em dia é dependente de dados. Isto significa que a forma como os dados estão organizados na memória secundária e a forma de se ter acesso a eles são ambas ditadas pelas necessidades da aplicação, e, além disso, o conhecimento da organização dos dados e técnicas de acesso está embutido na lógica da aplicação. Por exemplo, pode ser decidido que (por razões de desempenho) um determinado arquivo deva ser armazenado no formato indexado seqüencial. Então, a aplicação tem que saber que os índices existem e tem que conhecer a seqüência do arquivo (como definido pelo índice), sendo a estrutura interna da aplicação montada em torno deste conhecimento. Além disso, a forma precisa dos vários procedimentos de acesso e manuseio de exceções dentro da aplicação dependerá fundamentalmente do interface apresentado pelo software indexado seqüencial.

Dizemos que uma aplicação como esta é dependente dos dados, pois é impossível mudar a estrutura de armazenamento (como os dados estão fisicamente gravados) ou a estratégia de acesso (como ter acesso aos dados) sem afetar a aplicação, provavelmente de forma dramática. Por exemplo, não seria possível substituir o arquivo indexado seqüencial acima por um de endereçamento randômico sem efetuar grandes modificações na aplicação. É interessante notarmos, incidentalmente, que as porções da aplicação que exigiriam alterações nesse caso são as que se comunicam com o software de manuseio de arquivos, sendo que as dificuldades envolvidas são irrelevantes para o problema que a aplicação se destinou a resolver quando escrita — são dificuldades *introduzidas* pela estrutura do interface do software de manuseio de arquivos.

Entretanto, em um sistema de banco de dados seria extremamente indesejável permitir que as aplicações fossem dependentes de dados, por pelo menos duas razões.

1. Aplicações diferentes necessitarão visões diferentes do mesmo dado. Por exemplo, imaginemos que antes de ter a empresa introduzido seu banco de dados integrado, tivessemos duas aplicações, A e B, cada uma possuindo seu arquivo contendo o campo “saldo do cliente”. Imaginemos, também, que a aplicação A registra o valor em decimal, enquanto que a aplicação B o registra em binário. Ainda será possível integrar os dois arquivos e eliminar a redundância, desde que o DBMS execute todas as conversões necessárias entre a representação armazenada escolhida (que pode ser decimal, binária ou ainda outra) e a forma como cada aplicação deseja vê-la. Por exemplo, se a decisão for de manter o valor em decimal, então qualquer acesso de B exigirá uma conversão de ou para binário.

Este é um exemplo muito trivial do tipo de diferença que pode existir em um sistema de banco de dados entre a forma como os dados são vistos por uma aplicação e o que está realmente armazenado. Mais adiante iremos considerar muitas outras possíveis diferenças.

2. O DBA tem que ter liberdade para modificar a estrutura de armazenamento ou a estratégia de acesso (ou ambas) em resposta a mudanças de necessidades, sem ter que mudar as aplicações existentes. Por exemplo, a empresa pode adotar novos padrões; as prioridades das aplicações podem se modificar; novos tipos de dispositivos de armazenamento podem se tornar disponíveis; e assim por diante. Se as aplicações forem dependentes dos da-

dos, tais mudanças envolvem alterações correspondentes nos programas, prendendo esforço de programação que, de outra forma, estaria disponível para a criação de novas aplicações. Por exemplo, uma instalação de grande porte tem aproximadamente 25 por cento do seu esforço de programação dedicados a este tipo de atividade de manutenção [1.16]. Claramente, isto é um desperdício de recurso extremamente valioso.

Assim, a provisão de independência de dados é um objetivo maior dos sistemas de bancos de dados. Podemos definir independência de dados como a imunidade das aplicações a mudanças na estrutura de armazenamento ou na estratégia de acesso – o que naturalmente implica que as aplicações em foco não dependam de qualquer estrutura de armazenamento ou estratégia de acesso específicas. Na Seção 1.5, vamos apresentar uma arquitetura para um sistema de banco de dados que fornece a base para alcance desse objetivo. Antes disso, entretanto, vamos considerar em maiores detalhes os tipos de mudanças que o DBA pode desejar fazer (e às quais nós desejamos as aplicações imunes).

Vamos começar com algumas definições.

Um campo armazenado é a menor unidade de dados, com nome, armazenada no banco de dados. Em geral, o banco de dados conterá muitas ocorrências ou casos de cada um dos diversos tipos de campos armazenados. Por exemplo, um banco de dados contendo informações sobre peças provavelmente incluiria um tipo de campo armazenado chamado “número da peça”, e haveria uma ocorrência deste campo armazenado para cada peça distinta.²

Um registro armazenado é uma coleção, com nome, de campos associados armazenados. Novamente devemos fazer uma distinção entre tipo e ocorrência. Uma ocorrência ou caso de registro armazenado consiste de um grupo de ocorrências de campos armazenados (e representa uma associação entre eles). Por exemplo, uma ocorrência de registro armazenado poderia consistir de uma ocorrência de cada um dos seguintes campos armazenados: número da peça, nome da peça, cor da peça e peso da peça. Naturalmente a associação entre eles decorre do fato de representarem propriedades da mesma peça específica. Dizemos que o banco de dados contém múltiplas ocorrências do tipo de registro armazenado “peça” (novamente, uma ocorrência para cada peça distinta). Na maioria dos sistemas, a ocorrência do registro armazenado é a unidade de acesso ao banco de dados – isto é, a unidade que pode ser recuperada ou armazenada em um acesso pelo DBMS (veja Capítulo 2).

[Como um aparte, notemos que é comum retirar-se os qualificadores “tipo” e “ocorrência”, confiando no contexto para indicar o que se quer significar. Embora haja um pequeno risco de confusão, a prática é conveniente, e nós mesmos vamos adotá-la de vez em quando no que se segue.]

Um arquivo armazenado é uma coleção (com nome) de todas as ocorrências de um tipo de registro armazenado.³

Na maioria dos sistemas de hoje em dia, um registro lógico de aplicação é idêntico a um registro armazenado. No entanto, como nós já vimos, isto não ocorre necessariamente.

² “Peça” aqui refere-se à *espécie* de peça, digamos dobradiças (isto é, não a uma dobradiça específica).

³ Para simplificar, ignoramos a possibilidade de que um arquivo armazenado contenha mais de um tipo de registro armazenado. Esta simplificação não afetará materialmente nada na discussão subsequente.

te em um sistema de banco de dados, porque o DBA pode efetuar mudanças na estrutura armazenada — isto é, nos campos e registros armazenados — enquanto que os campos e registros lógicos correspondentes *não* se modificam. Por exemplo, o campo “peso da peça” mencionado acima poderia estar armazenado em binário para economizar espaço de armazenamento, enquanto que uma determinada aplicação COBOL poderia vê-lo como um item PICTURE (isto é, como uma seqüência de caracteres). Entretanto, uma diferença como essa — envolvendo conversão do tipo de dado em um campo específico a cada acesso — é comparativamente pequena; em geral, a diferença entre o que a aplicação vê e o que está realmente armazenado pode ser considerável. Este fato está ilustrado nas subseções seguintes descrevendo aspectos da estrutura de armazenamento do banco de dados que estão sujeitos a variação. (Uma lista mais completa pode ser encontrada em Engles [1.16].) O leitor deverá considerar em cada caso o que o DBMS teria que fazer para proteger uma aplicação de tal variação (e, sem dúvida, se tal proteção poderia sempre ser conseguida).

Representação de dados numéricos

Um campo numérico pode ser armazenado em forma aritmética interna (isto é, em decimal compactado) ou como uma seqüência de caracteres. Em cada caso o DBA tem que escolher apropriadamente: base (binária ou decimal), escala (fixa ou ponto flutuante), modo (real ou complexo), e precisão (número de dígitos). Qualquer destes aspectos pode ser modificado para melhorar o desempenho ou para adaptar-se a novos padrões ou por muitas outras razões.

Representação de dados caracteres

Um campo de seqüência de caracteres pode ser armazenado em qualquer dos diversos códigos de caracteres (por exemplo EBCDIC, ASCII).

Unidade dos dados numéricos

O sistema de unidade de um campo numérico pode sofrer modificação — de polegadas para centímetros, por exemplo.

Dados em código

Em algumas situações pode ser desejável representar o dado na memória através de valores em código. Por exemplo, o campo “cor da peça”, que a aplicação vê como uma seqüência de caracteres (‘VERMELHO’ ou ‘AZUL’ ou ‘VERDE’ . . .), pode estar armazenado como um único dígito decimal, interpretado de acordo com a tabela 1 = ‘VERMELHO’, 2 = ‘AZUL’, e assim por diante.

Materialização dos dados

Normalmente o campo lógico visto por uma aplicação corresponde a algum campo único armazenado (embora, como nós já vimos, possa haver diferenças no tipo de dado, unidades e assim por diante). Neste caso, o processo de materialização — isto é, a construção de uma ocorrência do campo lógico a partir da ocorrência correspondente do campo armazenado e sua apresentação à aplicação — pode ser considerado direto. Entretanto, ocasionalmente, um campo lógico não terá um único armazenado que lhe corresponda; ao invés disso, seus valores serão materializados por meio de alguma computação efetuada sobre,

um conjunto de diversas ocorrências de campos armazenados. Por exemplo, valores do campo lógico "quantidade total" podem ser materializados pela soma de diversos valores de quantidades individuais. "Quantidade total" aqui é um exemplo de um campo *virtual*, e o processo de materialização é dito ser indireto. (Note, entretanto, que o usuário pode ver uma diferença entre campos reais e virtuais, quando mais não seja porque provavelmente não seria possível criar ou modificar uma ocorrência de um campo virtual, pelo menos não diretamente.)

Estrutura dos registros armazenados

Dois tipos existentes de registros armazenados podem ser combinados em um. Por exemplo, os tipos de registros (números da peça, cor) e (número da peça, peso) podem ser integrados para dar (número da peça, cor, peso). Isto ocorre comumente quando aplicações anteriores ao banco de dados são trazidas para o sistema de banco de dados. Isto implica que o registro lógico de uma aplicação pode consistir de algum subconjunto de um registro armazenado (isto é, parte dos campos armazenados podem estar invisíveis a uma aplicação específica).

Alternativamente, um tipo de registro armazenado único pode ser dividido em dois. Por exemplo (número da peça, cor, peso) pode ser repartido em (número da peça, cor) e (número da peça, peso). Esta divisão poderia permitir, por exemplo, que partes menos utilizadas fossem armazenadas em dispositivos mais lentos. A implicação aqui é que o registro lógico de uma aplicação pode conter campos de diversos registros armazenados.

Estrutura dos arquivos armazenados

Um determinado arquivo armazenado pode estar fisicamente implementado em uma grande variedade de formas. Por exemplo, ele pode estar inteiramente contido dentro de um volume de armazenamento (por exemplo, um volume de disco), ou pode estar distribuído em diversos volumes de vários tipos diferentes. Ele pode ou não encontrar-se organizado seqüencialmente de acordo com os valores de algum campo armazenado. Ele pode ou não estar organizado seqüencialmente de alguma outra forma, por exemplo, por um ou mais índices associados ou por meio de indicadores de localização embutidos. A ele se poderá ou não ter acesso via endereçamento randômico. Os registros armazenados poderão estar blocados ou não. Mas nenhuma destas considerações deve afetar as aplicações sob qualquer aspecto (a não ser, naturalmente, em desempenho).

Fica implícito na lista acima que o banco de dados seja (ou deva ser) capaz de crescer sem afetar as aplicações existentes. Sem dúvida, uma das maiores razões para a provisão de independência de dados é permitir que o banco de dados cresça sem impactar as aplicações existentes. Por exemplo, deve ser possível estender um tipo de registro armazenado pela adição de novos tipos de campos (tipicamente representando informações adicionais relativas a algum tipo existente de entidade ou relacionamento; por exemplo, um campo "custo unitário" poderia ser acrescentado ao registro armazenado "peças"). Estes novos campos ficarão simplesmente invisíveis a todas as aplicações anteriores. Semelhantemente, deve ser possível adicionar-se tipos inteiramente novos de registros armazenados ao banco de dados, novamente sem exigir qualquer mudança nas aplicações existentes; tipicamente, tais registros representariam tipos completamente novos de entidades ou relacionamentos (por exemplo, um registro de "fornecedor" poderia ser adicionado ao banco de dados.)

1.5 UMA ARQUITETURA PARA SISTEMAS DE BANCO DE DADOS

Estamos agora em posição para esboçar uma arquitetura para um sistema de banco de dados (Figs. 1.3, 1.4, e 1.5). Nosso propósito ao apresentar esta arquitetura é estabelecer uma base sobre a qual possamos evoluir nos capítulos subsequentes. Esta estruturação será extremamente útil para descrevermos os conceitos gerais de bancos de dados e para explicarmos a estrutura dos sistemas específicos; mas não queremos dizer com isto que todos os sistemas de bancos de dados se encaixem perfeitamente nesta estruturação específica, nem queremos sugerir que esta arquitetura particular estabelece a única base possível. Entretanto, a arquitetura parece se ajustar razoavelmente bem a um grande número de sistemas; além disso, ela está em larga concordância com a que foi proposta pelo ANSI/SPARC Study Group on Data Base Management Systems [1.17, 1.18]. (No entanto, nós decidimos não seguir a terminologia ANSI/SPARC em todos os detalhes.)

A arquitetura está dividida em três níveis gerais: interno, conceitual e externo (Fig. 1.3). Grosso modo, o nível interno é o mais próximo do armazenamento físico, isto é, o que está voltado para a forma como os dados estão realmente armazenados; o nível externo é o mais próximo dos usuários, isto é, o que está voltado para a forma como os dados são vistos por cada usuário; e o nível conceitual é um “nível de simulação” entre os outros dois. Se o nível externo está voltado para as visões de *cada* usuário, o nível conceitual pode ser imaginado como definindo a visão da comunidade de usuários. Em outras palavras, haverá muitas “visões externas”, cada uma consistindo de uma representação mais ou menos abstrata de alguma porção do banco de dados, e haverá uma única “visão conceitual”, correspondendo a uma representação semelhantemente abstrata representando o banco de dados na sua totalidade.⁴ (Lembre-se de que a maioria dos usuários não estará interessada no total do banco de dados, mas somente em uma porção restrita dele.) Da mesma forma, haverá uma única “visão interna”, representando o total do banco de dados como ele está realmente armazenado.

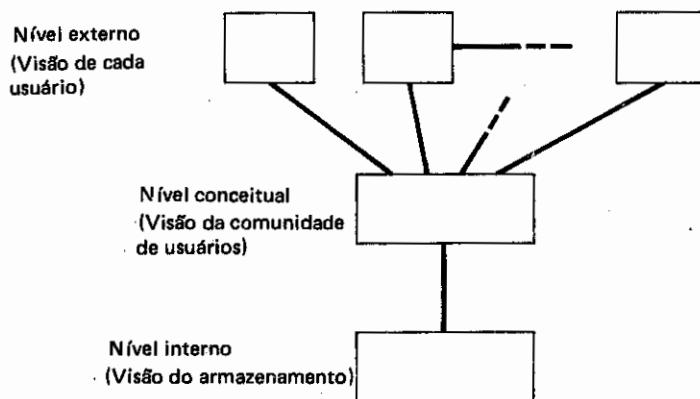


Fig. 1.3 Os três níveis da arquitetura

4

Quando descrevemos alguma representação como abstrata, queremos significar simplesmente que ela envolve formações voltadas para o usuário (tais como registros e campos lógicos), ao invés de formações voltadas para a máquina (tais como bits e bytes).

Um exemplo irá ajudar a tornar mais claras estas idéias. A Fig. 1.4 mostra a estrutura conceitual de um banco de dados simples de pessoal, a estrutura interna e duas estruturas externas correspondentes (uma para um usuário de PL/I e outra para um usuário de COBOL). Naturalmente, o exemplo é completamente hipotético – não pretende parecer-se com nenhum sistema real -- e foram deliberadamente omitidos muitos detalhes irrelevantes.

Devemos interpretar a Fig. 1.4 como se segue.

- No nível conceitual, o banco de dados contém informações pertinentes ao tipo de entidade chamada EMPLOYEE. Cada EMPLOYEE possui um EMPLOYEE_NUMBER (seis caracteres), um DEPARTMENT NUMBER (quatro caracteres), e um SALARY (cinco dígitos).
- No nível interno, os empregados estão representados por um tipo de registro armazenado chamado STORED_EMP, com dezoito bytes de comprimento. STORED_EMP contém quatro tipos de campos armazenados: um prefixo de seis bytes (contendo presumivelmente informações de controle tais como *flags* ou indicadores de localização) e três campos de dados correspondendo às três propriedades de EMPLOYEE. Em adição, os registros STORED_EMP estão indexados no campo EMP# por um índice chamado EMPX.
- O usuário PL/I tem uma visão externa do banco de dados na qual cada empregado está representado por um registro PL/I contendo dois campos (os números de departamento não interessam a este usuário e foram omitidos da visão). O tipo de registro está definido por uma declaração comum de estrutura do PL/I de acordo com as regras normais do PL/I.

Externo (PL/I)	Externo (COBOL)
DCL 1 EMPP, 2 EMP# CHAR(6), 2 SAL FIXED BIN(31);	01 EMPC. 02 EMPNO PIC X(6). 02 DEPTNO PIC X(4).
Conceitual	
EMPLOYEE EMPLOYEE_NUMBER CHARACTER (6) DEPARTMENT_NUMBER CHARACTER (4) SALARY NUMERIC (5)	
Interno	
STORED_EMP LENGTH=18 PREFIX TYPE=BYTE(6),OFFSET=0 EMP# TYPE=BYTE(6),OFFSET=6,INDEX=EMPX DEPT# TYPE=BYTE(4),OFFSET=12 PAY TYPE=FULLWORD,OFFSET=16	

Fig. 1.4 Um exemplo dos três níveis

- Semelhantemente, o usuário COBOL tem uma visão externa na qual cada empregado está representado por um registro COBOL contendo, novamente, dois campos (foi omitido o de salário). O tipo de registro está definido por uma descrição de registro comum do COBOL de acordo com as regras normais do COBOL.
- Note que objetos correspondentes nos três níveis podem ter nomes diferentes em cada nível. Por exemplo, o campo de número de empregado no COBOL é chamado de EMPNO, o campo armazenado correspondente é chamado de EMP#, e o objeto conceitual correspondente é chamado de EMPLOYEE_NUMBER. (O sistema tem que estar ao par das correspondências. Por exemplo, ele tem que ser informado de que o campo EMPNO no COBOL é derivado do objeto conceitual EMPLOYEE_NUMBER, que por sua vez é representado pelo campo armazenado EMP#. Tais correspondências, ou *mapeamentos*, não estão mostrados na Fig. 1.4.)

Vamos agora examinar os componentes da arquitetura com mais detalhes (Fig. 1.5).

Os usuários são tanto programadores de aplicações como usuários de terminais *on-line* de qualquer grau de sofisticação. (O DBA é um importante caso especial.) Cada usuário dispõe de uma *linguagem* para seu uso. Para o programador de aplicações deverá ser uma linguagem convencional de programação, como o COBOL ou o PL/I; para o usuário de terminal deverá ser ou uma linguagem de consulta ou uma linguagem com finalidades específicas modelada segundo as necessidades do usuário e apoiada por um programa de aplicação *on-line*. Para nossas finalidades, o fato importante sobre linguagem do usuário é o de que ela deve incluir uma *sublinguagem de dados* (DSL), isto é, um subconjunto da linguagem total voltada para os objetos e operações do banco de dados. Estamos considerando a sublinguagem de dados embutida em uma *linguagem principal*. Um determinado sistema pode suportar várias linguagens principais e várias sublinguagens de dados.

Em princípio, uma dada sublinguagem de dados compõe-se realmente da combinação de duas linguagens: uma linguagem de definição de dados (DDL), que possibilita a definição ou descrição dos objetos do banco de dados (como eles são percebidos pelo usuário), e uma linguagem de manipulação de dados (DML), que suporta a manipulação ou processamento desses objetos. Consideremos o usuário PL/I da Fig. 1.4. A sublinguagem de dados para aquele usuário consiste dos dispositivos do PL/I que são utilizados para a comunicação com o banco de dados. A porção DDL consiste daquelas formações declarativas do PL/I que são necessárias para declarar os objetos do banco de dados: a própria instrução DECLARE (DCL), alguns tipos de dados PL/I, possivelmente extensões especiais ao PL/I para suportar novos objetos que não sejam manuseados pelo PL/I existente. (A versão corrente do PL/I [1.27] realmente não inclui qualquer dispositivo específico para bancos de dados.) A porção DML consiste das instruções executáveis do PL/I que transferem informações de e para o banco de dados — novamente, possivelmente incluindo novas instruções especiais.

A explicação anterior parte do princípio de que a sublinguagem de dados e a principal (PL/I no exemplo) estão muito “estreitamente unidas” — isto é, para o usuário as duas realmente não são separáveis. Na prática corrente este não é o caso usual, pelo menos no que tange a linguagens de *programação*. Ao invés disso, (a) as definições ficam totalmente fora do programa de aplicação, e são escritas em uma DDL que nem de leve se parece com a linguagem principal do usuário, e (b) a manipulação é feita pela chamada (CALL) de sub-rotinas padrão (fornecidas como parte do DBMS), estando portanto também fora da estrutura da linguagem principal. Em outras palavras, na maioria dos sistemas

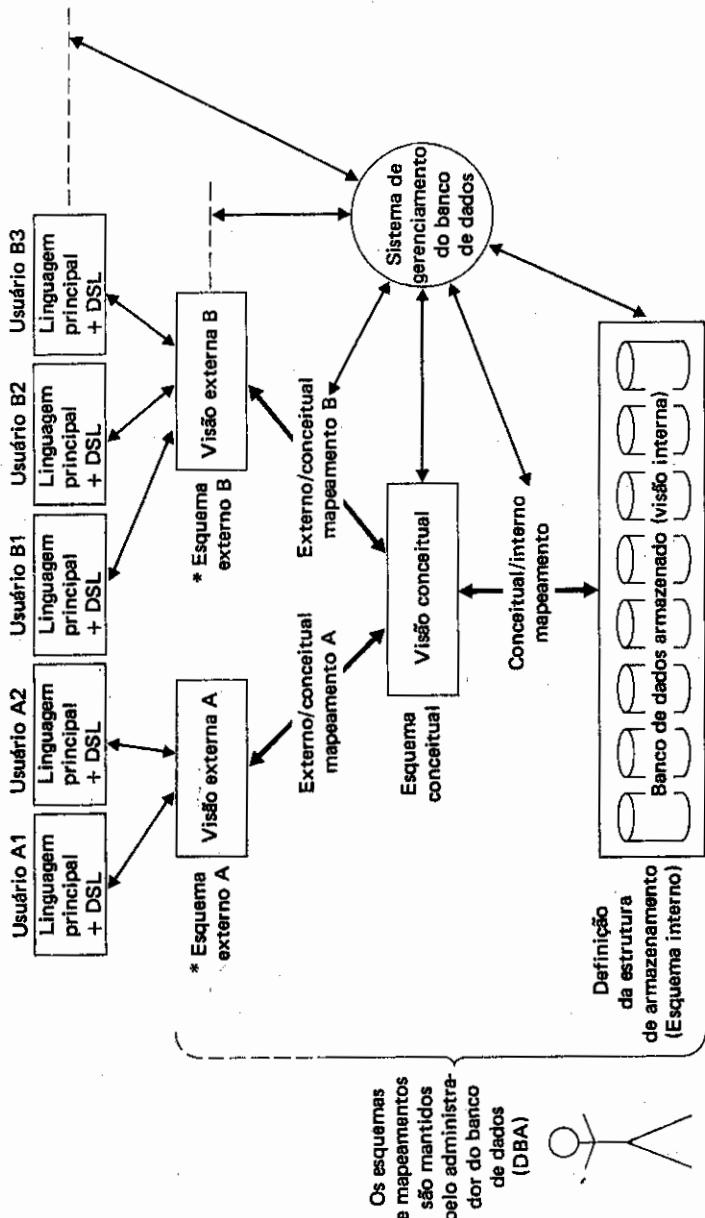


Fig. 1.5 Arquitetura de um sistema de banco de dados

de hoje a sublinguagem de dados e a principal estão *frouxamente* unidas. Um sistema estreitamente unido fornece um conjunto de facilidades mais uniforme para o usuário, mas obviamente requer mais esforço por parte dos que projetam e desenvolvem tais sistemas (o que é uma razão para a situação atual). Parece provável que venha a ocorrer um movimento no sentido de sistemas estreitamente unidos no decorrer dos próximos anos.

Retornando à arquitetura: Já mencionamos que um usuário isolado estará geralmente interessado somente em alguma parcela do banco de dados total; além disso, a visão que o usuário tem daquela parcela é em geral algo abstrata quando comparada com a forma como os dados estão fisicamente armazenados. Em termos ANSI/SPARC a visão de um determinado usuário é denominada de *visão externa*.⁵ Uma visão externa é portanto o conteúdo do banco de dados visto por um determinado usuário (isto é, para aquele usuário, a visão externa é o banco de dados). Por exemplo, um usuário do departamento do pessoal pode ver o banco de dados como sendo uma coleção de ocorrências de registros de departamentos mais uma coleção de ocorrências de registros de empregados (podendo estar totalmente desinformado sobre as ocorrências de registros de fornecedores e peças vistas pelos usuários do departamento de compras). Em geral, portanto, uma visão externa consiste de múltiplas ocorrências de múltiplos tipos do *registro externo*.⁶ Um registro externo *não* é necessariamente o mesmo que um registro armazenado. A sublinguagem de dados do usuário é definida em termos de registros externos; por exemplo, uma operação “*get*” da DML irá recuperar uma ocorrência de registro externo, ao invés de uma ocorrência de registro armazenado. (Agora nós podemos ver, incidentalmente, que o termo “registro lógico” usado na seção 1.4 realmente se refere a um registro externo. A partir deste ponto, nós geralmente evitaremos o termo “registro lógico.”)

Cada visão externa é definida por meio de um *esquema externo*, que consiste basicamente de definições de cada um dos vários tipos de registro externo naquela visão externa. (O esquema externo é escrito usando-se a porção DDL da sublinguagem de dados. Essa DDL é por isso algumas vezes chamada de *DDL externa*.) Por exemplo, o tipo de registro externo empregado pode ser definido como um campo de seis caracteres ‘employee-number’ mais um campo binário fixo ‘salary’, e assim por diante. Além disso tem que haver uma definição do *mapeamento* entre o esquema externo e o esquema conceitual básico (descrito abaixo). Mais tarde iremos discutir esse mapeamento.

Vamos observar agora o nível conceitual. A *visão conceitual* é uma representação do conteúdo completo do banco de dados, novamente em uma forma que é algo abstrata quando comparada com a forma como os dados estão fisicamente armazenados. (Podendo também ser bastante diferente da forma como os dados são vistos por um usuário em particular. *Grosso modo*, podemos dizer que esta é uma visão dos dados “como eles realmente são”, e não como os usuários são forçados a vê-los devido a restrições [por exem-

⁵ Visões externas foram chamadas de submodelos de dados na primeira edição deste livro, e de modelos externos (de acordo com [1.17] na segunda. Há muita confusão sobre terminologia nesta área. Acompanhando [1.18], nós agora preferimos “visão” para um conjunto de ocorrências e “esquema” para a definição de uma visão.

⁶ Neste momento vamos supor que toda a informação está representada sob a forma de registros. Mais tarde vamos ver que a informação pode também estar representada de outras formas, por exemplo, na forma de “interligações” (veja o Capítulo 3). Para um sistema que use tais métodos alternativos, as definições e explicações desta seção precisarão ser adequadamente modificadas.

plo] da linguagem ou do *hardware* específicos que eles estejam usando.) A visão conceitual consiste de múltiplas ocorrências de múltiplos tipos de *registros conceituais*⁷; por exemplo, ela pode consistir de uma coleção de ocorrências de registros de departamentos mais uma coleção de ocorrências de registros de empregados mais uma coleção de ocorrências de registros de fornecedores, mais uma coleção de ocorrências de registros de peças... Um registro conceitual não é o mesmo, necessariamente, que um registro externo por um lado, nem que um registro armazenado por outro. A visão conceitual é definida por meio de um *esquema conceitual*, que inclui definições de cada um dos vários tipos de registros conceituais. (O esquema conceitual é escrito usando-se outra linguagem de definição de dados – a *DDL conceitual*.) Para que se possa manter a independência de dados, essas definições não podem envolver quaisquer considerações sobre estrutura de armazenamento ou estratégia de acesso – elas têm que ser *somente* definições do conteúdo da informação. Portanto não podem existir referências a representações do campo armazenando, seqüência física, indexação, endereçamento randômico, ou outros detalhes referentes a armazenamento/acesso. Se o esquema conceitual for realmente feito independente de dados desta maneira, os esquemas externos, que são definidos em termos do esquema conceitual (veja abaixo), serão por consequência também independentes dos dados.

A visão conceitual, então, é uma visão do conteúdo total do banco de dados, e o esquema conceitual é uma definição dessa visão. Seria entretanto enganoso sugerir que o esquema conceitual não seja mais do que um conjunto de definições parecidas com as simples definições de registros de um programa COBOL atual. As definições no esquema conceitual devem incluir uma grande quantidade de dispositivos adicionais, tais como autorizações de utilização e procedimentos de validação mencionados na Seção 1.3. Algumas autoridades vão ainda mais longe, insinuando que o objetivo final do esquema conceitual é descrever toda a empresa – não somente seus dados operacionais, mas também como esses dados são usados: como os dados fluem de ponto para ponto dentro da empresa, para que o dado é usado naquele ponto, que controles de auditoria devem ser aplicados em cada ponto, e assim por diante. No entanto, devemos enfatizar que poucos sistemas atuais – se é que algum – suportam um nível conceitual que se aproxime a tal grau de amplitude; na maioria dos sistemas existentes, a visão conceitual é realmente pouco mais do que uma simples união das visões individuais de todos os usuários, com a provável adição de alguns procedimentos simples de autorização e validação. Mas tudo indica que os futuros sistemas de banco de dados serão eventualmente bem mais sofisticados em seu suporte ao nível conceitual. Nós vamos discutir este tópico em maior profundidade na parte final deste livro.

O terceiro nível da arquitetura é o nível interno. A *visão interna* é uma representação de nível bastante baixo de todo o banco de dados; ela consiste de múltiplas ocorrências de múltiplos tipos de *registros internos*.⁸ “Registro interno” é o termo ANSI/SPARC para a formação que nós temos chamado de *registro armazenado* (e de forma geral nós

⁷ A nota de rodapé 6 também se aplica ao nível conceitual. Deve também ser enfatizado que pode haver outras maneiras preferíveis para a modelagem dos dados operacionais da empresa no nível conceitual. Por exemplo, ao invés de trabalhar em termos de “registros conceituais”, pode ser mais desejável considerar “entidades”, e talvez também “relacionamentos”, de uma forma algo mais direta. No entanto, estas questões estão além do escopo deste capítulo introdutório.

⁸ A nota de rodapé 6 também se aplica ao nível interno.

continuaremos a usar este último termo); a visão interna está portanto logo acima do nível físico, uma vez que ela não lida em termos de registros *físicos* ou blocos, nem com restrições específicas de dispositivos tais como tamanho de cilindro ou trilha. (Basicamente a visão interna assume um espaço infinito linear de endereçamento. Detalhes de como este espaço de endereçamento é mapeado em relação à memória física são altamente específicos em cada implementação, e não serão explicitamente abordados na arquitetura.) A visão interna é descrita por meio do *esquema interno*, que não somente define os vários tipos de registros armazenados, mas também especifica que índices existem, como os campos armazenados estão representados, qual a sequência física dos registros armazenados, e assim por diante. (O esquema interno é preparado usando-se ainda outra linguagem de definição de dados – a *DDL interna*.) Neste livro nós tenderemos a usar o termo “banco de dados armazenado” no lugar de “visão interna”, e “definição da estrutura de armazenamento” no lugar de “esquema interno”. Usaremos também em geral (mas não invariavelmente!) o termo sem qualificativo “banco de dados” como um sinônimo de “banco de dados armazenado”, reservando-o portanto para significar o que está realmente armazenado; mas chamo a atenção do leitor para o fato de que esta interpretação não está absolutamente universalizada.⁹

Fazendo nova referência à figura 1.5, o leitor poderá observar dois níveis de *mapeamento*, um entre os níveis externo e conceitual do sistema e outro entre os níveis conceitual e interno. O *mapeamento conceitual/interno* define a correspondência entre a visão conceitual e o banco de dados armazenado; ele especifica como os registros e campos conceituais são mapeados em relação aos seus correspondentes armazenados. Se a estrutura do banco de dados armazenado for modificada — isto é, se for feita uma modificação na definição da estrutura armazenada —, o mapeamento conceitual/interno tem que ser alterado de acordo, para que o esquema conceitual possa permanecer invariável. (É responsabilidade do DBA controlar essas mudanças.) Em outras palavras, os efeitos dessas mudanças têm que ser contidos abaixo do nível conceitual, de forma a se alcançar a independência de dados.

Um *mapeamento externo/conceitual* define a correspondência entre uma determinada visão externa e a visão conceitual. Em geral, podem existir entre esses dois níveis os mesmos tipos de diferenças que podem ocorrer entre a visão conceitual e o banco de dados armazenado. Por exemplo, os campos podem ter tipos de dados diferentes, os registros podem estar sequenciados diferentemente, e assim por diante. Pode existir qualquer número de visões externas ao mesmo tempo; qualquer quantidade de usuários pode compartilhar uma dada visão externa; diferentes visões externas podem se sobrepor. Incidentalmente, alguns sistemas permitem que se defina uma visão externa em termos de outras, aos invés de sempre exigirem uma definição explícita do mapeamento relativo ao nível conceitual (um dispositivo muito útil quando existem diversas visões externas muito parecidas umas com as outras).

⁹

Em situações excepcionais, os programas de aplicação podem operar diretamente no nível interno ao invés de no nível externo. Nem precisamos dizer que esta prática não é recomendada; ela representa um risco de segurança (pois as verificações de segurança são ignoradas), e um risco de integridade (pois os procedimentos de validação são ignorados), além dos programas se tornarem dependentes dos dados; mas em algumas ocasiões esta é a única forma de se obter a função ou a *performance* requeridas — assim como um programador COBOL pode ocasionalmente precisar se utilizar da linguagem Assembler para satisfazer hoje a certos objetivos funcionais ou de *performance*.

Voltando novamente à figura 1.5, podemos ver que ainda restam três tópicos a serem discutidos: o sistema de gerenciamento do banco de dados, o administrador do banco de dados, e o interface com o usuário. O *sistema de gerenciamento do banco de dados* (DBMS) é o *software* que manipula todos os acessos ao banco de dados. Conceitualmente, o que ocorre é o seguinte: (1) Um usuário emite uma solicitação, usando alguma determinada linguagem de manipulação de dados; (2) o DBMS intercepta a solicitação e a interpreta; (3) o DBMS, por seu lado, consulta o esquema externo, o mapeamento externo/conceitual, e a definição da estrutura de armazenamento; e (4) o DBMS executa as operações necessárias no banco de dados armazenado. Por exemplo, consideremos o que está envolvido na recuperação de uma determinada ocorrência de registro externo. Em geral, serão necessários campos de diversas ocorrências de registros conceituais. Cada ocorrência de registro conceitual, por seu turno, pode necessitar de campos de diversas ocorrências de registros armazenados. Então, pelo menos conceitualmente, o DBMS tem que recuperar todas as ocorrências requeridas de registros armazenados, construir as necessárias ocorrências de registros conceituais, e então construir as ocorrências requeridas de registros externos. Em cada etapa podem ser necessárias conversões de tipo de dado ou outras.

(A discussão anterior parte do princípio de que todo o processo é interpretativo, o que normalmente implica em um desempenho bastante ruim. Naturalmente na prática, é possível algumas vezes *compilar* antecipadamente as solicitações de acesso, evitando a sobrecarga interpretativa.)

Retornando à Fig. 1.5: O *administrador do banco de dados* (DBA), sobre o qual já fizemos alguma exposição, é a pessoa (ou grupo de pessoas) responsável pelo controle global do banco de dados. As responsabilidades do DBA incluem o seguinte.

Responsabilidades do DBA:

Decidir qual o conteúdo de informações do banco de dados

É parte do trabalho do DBA decidir exatamente qual informação deve ser mantida no banco de dados – em outras palavras, identificar as entidades que interessam à empresa e a informação a ser registrada sobre essas entidades. Uma vez isto feito, o DBA tem então que definir o conteúdo do banco de dados escrevendo o esquema conceitual (usando a linguagem de definição de dados conceitual). Este esquema em formato objeto (compilado) é usado pelo DBMS ao responder às solicitações de acesso. O formato fonte (não-compilado) serve como um documento para referência dos usuários do sistema.

Decidir qual a estrutura de armazenagem e a estratégia de acesso

O DBA tem que decidir também como os dados deverão estar representados no banco de dados, e precisa especificar esta representação escrevendo a definição da estrutura de dados (usando a linguagem de definição interna de dados). Além disso, têm também que ser especificados os mapeamentos associados entre a definição da estrutura de dados e o esquema conceitual. Na prática, provavelmente a DDL interna ou a DDL conceitual incluirão os meios para especificação desse mapeamento, mas as diferentes definições devem estar claramente separadas. Da mesma forma que com o esquema conceitual, o esquema interno e o mapeamento correspondente existirão tanto no formato fonte como no formato objeto.

Ser o elo de ligação com os usuários

É função do DBA servir como elemento de ligação com os usuários, para garantir a disponibilidade dos dados de que estes necessitam, e para preparar os esquemas externos que

sejam necessários (usando a linguagem de definição de dados externos apropriada; como já foi mencionado, podem existir diversas DDLs externas distintas). Além disso, têm que ser especificados os mapeamentos entre os esquemas externos e o esquema conceitual. Na prática, a DDL externa provavelmente incluirá os meios para a especificação desse mapeamento, mas deve haver uma clara distinção entre o esquema e o mapeamento. Cada esquema externo e seu mapeamento correspondente existirá tanto no formato fonte como no formato objeto.

Definir as verificações de autorização e os procedimentos de validação

Como já mencionado, as verificações de autorização e os procedimentos de validação podem ser considerados como extensões lógicas do esquema conceitual. A DDL conceitual deverá apresentar meios para a especificação de tais verificações e procedimentos.

Definir uma estratégia para *back-up* e recuperação

A partir do momento em que a empresa começa efetivamente a se basear em bancos de dados, ela se torna dependente de forma crítica da operação bem-sucedida desse sistema. Caso ocorra avaria a alguma porção do banco de dados — causada por erro humano, digamos, ou por falha no *hardware* ou no sistema operacional de suporte —, é essencial que se possam reparar os dados envolvidos com um mínimo de atraso, bem como causar o menor efeito possível ao restante do sistema. (Por exemplo, não deve ser afetada de nenhuma forma a disponibilidade dos dados que *não* sofreram avaria.) O DBA tem que definir e implementar uma estratégia apropriada de recuperação, envolvendo, por exemplo, a descarga periódica do banco de dados para uma fita de *back-up* e procedimentos para re-carregar porções relevantes do banco de dados a partir dessa fita.

Monitorar o desempenho e responder a mudanças de necessidades

O DBA é responsável tanto pela organização do sistema como por obter o desempenho que seja “o melhor para a empresa”, e por fazer os ajustes necessários quando há mudanças de requisitos. Como já mencionado, quaisquer mudanças nos detalhes de armazenamento e acesso têm que ser seguidas por mudanças correspondentes na definição do mapeamento do armazenamento, de tal forma que o esquema conceitual possa permanecer constante.

É claro que o DBA irá precisar de diversos programas utilitários para ajudá-lo em suas tarefas. Esses utilitários devem ser uma parte essencial de um sistema prático de banco de dados, embora não tenham sido mostrados na Fig. 1.5. Abaixo encontramos alguns exemplos dos tipos de utilitários necessários.

- Rotinas de carga (para criar a versão inicial do banco de dados).
- Rotinas de reorganização (para, por exemplo, rearrumar o banco de dados e reutilizar espaço ocupado por dados obsoletos).
- Rotinas de controle de uso (para anotar cada operação feita no banco de dados, juntamente com a informação sobre o usuário que a realizou e os registros dos estados anterior e posterior).
- Rotinas de recuperação (para restaurar o banco de dados a um estado anterior depois de uma falha de *hardware* ou de programação).
- Rotinas de análise estatística (para ajudá-lo na monitoração do desempenho).

Os programas utilitários podem ser vistos como aplicações especiais fornecidas com o sistema (exceto as rotinas de controle de uso, que têm que ser parte do próprio DBMS central). Alguns utilitários operam diretamente ao nível interno (veja nota de rodapé 9).

Uma das ferramentas mais importantes para o DBA é o *dicionário de dados* (não mostrado na Fig. 1.5.). O dicionário de dados, por si mesmo, é efetivamente um banco de dados — um banco de dados que contém “dados sobre dados” (isto é, descrições de outros objetos no sistema ao invés de simplesmente “dados em bruto”). Em particular, encontram-se fisicamente armazenados no dicionário, tanto no formato fonte como no objeto, todos os vários esquemas (externo, conceitual, interno). Um dicionário completo deve também incluir informações de referências cruzadas mostrando, por exemplo, que programas utilizam que porções de dados, que departamentos necessitam que relatórios, e assim por diante. De fato, o dicionário pode até estar integrado no banco de dados por ele descrito, incluindo portanto a sua própria descrição. Deve ser possível consultar-se o dicionário da mesma forma que qualquer outro banco de dados, de tal maneira que o DBA possa, por exemplo, descobrir facilmente que programas poderão ser afetados por alguma mudança proposta ao sistema.

O último componente na Fig. 1.5 é o *interface com o usuário*. Isto pode ser definido como a fronteira do sistema além da qual tudo se torna invisível ao usuário. Por definição, portanto, o interface com o usuário encontra-se no nível *externo*. Entretanto, como veremos mais tarde (no Capítulo 9), há algumas situações nas quais a visão externa não pode diferir significativamente da porção relevante da visão conceitual básica.

1.6 BANCOS DE DADOS DISTRIBUÍDOS

Vamos concluir este capítulo com uma breve menção sobre o tópico *bancos de dados distribuídos*. A tecnologia dos bancos de dados distribuídos é um desenvolvimento comparativamente recente no campo global dos bancos de dados [1.26]. Um banco de dados distribuído é tipicamente aquele que não é inteiramente armazenado em uma única localização física, estando disperso através de uma rede de computadores geograficamente afastados e conectados por elos de comunicações. Como um exemplo (muito simplificado), consideremos o sistema de um banco no qual o banco de dados das contas dos clientes esteja distribuído pelas agências desse banco, de tal forma que cada registro individual de conta de cliente se encontre armazenado na agência local do cliente. Em outras palavras, o dado esteja armazenado no local no qual é mais freqüentemente usado, mas ainda assim disponível, via rede de comunicações, a usuários de outros locais (por exemplo, usuários da agência central do banco). As vantagens dessa distribuição são claras: combinam a eficiência do processamento local (sem sobrecarga de comunicações) na maioria das operações, com todas as vantagens discutidas anteriormente (em particular, o compartilhamento dos dados) apresentadas por um sistema centralizado. Mas, naturalmente, também há desvantagens: podem ocorrer elevadas sobrecargas nas comunicações, além de dificuldades técnicas significativas para se implementar esse sistema.

Um objetivo primordial em um sistema distribuído é o de que ele *pareça ser, ao usuário, um sistema centralizado*. Isto é, normalmente o usuário não precisará saber onde se encontra fisicamente armazenada determinada porção dos dados (ficando as aplicações independentes da maneira pela qual os dados são distribuídos, tornando possível uma mudança de distribuição sem que as aplicações sejam afetadas — outro aspecto da independência de dados). Portanto o fato de ser o banco de dados distribuído só deve ser relevante ao nível interno, e não aos níveis externo e conceitual. Poucos sistemas hoje em dia, se algum, avançaram muito na direção deste objetivo.

EXERCÍCIOS

O material apresentado neste capítulo não apresenta dificuldade, mas envolve uma massa desconcertante de terminologias. Para garantir que o leitor tenha absorvido as idéias mais importantes, pedimos que procure responder às seguintes perguntas.

- 1.1 Desenhe um diagrama da arquitetura de um sistema de bancos de dados apresentado na Seção 1.5.
- 1.2 Defina os seguintes termos.
 - banco de dados: integrado, compartilhado
 - esquema externo, visão, DDL
 - sistema de bancos de dados
 - esquema conceitual, visão, DDL
 - DBMS
 - esquema interno, visão, DDL
 - DBA
 - mapeamento externo/conceitual
 - usuário
 - mapeamento conceitual/interno
 - entidade
 - sublinguagem de dados
 - relacionamentos
 - dicionário de dados
 - independência de dados
 - segurança
 - arquivo, registro e campo armazenados
 - integridade
 - DML
- 1.3 Quais são as vantagens de se usar um sistema de banco de dados?
- 1.4 Quais são as desvantagens de se usar um sistema de banco de dados?

REFERÊNCIAS E BIBLIOGRAFIA

- 1.1 CODASYL Systems Committee. "A Survey of Generalized Data Base Management Systems". Technical Report (maio de 1969). Disponível na ACM e na IAG.
- 1.2 CODASYL Systems Committee. "Feature Analysis of Generalized Data Base Management Systems". Artigo técnico (maio de 1971). Disponível nas ACM, BCS e IAG.

Esses dois longos documentos (mais de 900 páginas juntos) complementam-se no seguinte sentido: [1.1] consiste de descrições independentes de vários sistemas (isto é, há um capítulo separado dedicado a cada sistema); [1.2] consiste de comparações dispositivo a dispositivo de um (ligeiramente diferente) conjunto de sistemas. Estão cobertos os seguintes sistemas: ADAM ([1.1] somente), COBOL ([1.2] somente), DBTG ([1.2] somente), GIS, IDS, ISL-1([1.1] somente), IMS ([1.2] somente), MARK IV, NIPS/FFS, SC-1, TDMS, UL/1.
- 1.3 CODASYL Systems Committee. "Introduction to Feature Analysis of Generalized Data Base Management Systems". *Comp. Bull.* 15, nº 4 (abril de 1971); também *CACM* 14, nº 5 (maio de 1971).

Adaptado da seção inicial de [1.2]. Uma introdução útil a alguns dos conceitos básicos sobre um sistema de banco de dados.
- 1.4 GUIDE/Sshare Data Base Task Force. "Data Base Management System Requirements." (novembro de 1971). Disponível na SHARE Inc., 111E Wacker Drive, Chicago, Illinois 60601.

Um documento detalhado preparado por representantes das associações de usuários IBM sobre os dispositivos requeridos por um sistema de banco de dados ideal.
- 1.5 R. G. Canning. "Trends in Data Management". Part 1, *EDP Analyzer* 9, nº 5 (maio de 1971); Part 2, *EDP Analyzer* 9, nº 6 (junho de 1971).

Essas duas edições do EDP Analyzer contêm descrições básicas sobre os seguintes sistemas: TOTAL, DBOMP, MARS III, CZAR, Disk Forte, SERIES, IMS (Versão 2), e dataBASIC. Em cada caso estão incluídas notas sobre algumas experiências reais dos usuários. Está também incluído um breve estudo sobre banco de dados.

- 1.6 G. C. Everest. *Data Base Management: Objectives, System Functions, and Administration*. New York: McGraw-Hill (1977).
- 1.7 D. C. Tsichritzis e F. H. Lochovsky. *Data Base Management Systems*. New York: Academic Press (1977).
Inclui descrições do IMS, System 2000, IDMS, TOTAL, e ADABAS.
- 1.8 D. Kroenke. *Database Processing: Fundamentals, Modeling, Applications*. Palo Alto, Calif.: Science Research Associates (1977).
Inclui descrições de ADABAS, System 2000, TOTAL, IDMS, IMS e MAGNUM.
- 1.9 R. G. Ross. *Data Base Systems: Design, Implementation, and Management*. New York: Amacom (1978).
Inclui resumos muito breves sobre 20 sistemas implementados.
- 1.10 A. F. Cardenas. *Data Base Management Systems*. Boston: Allyn and Bacon (1979).
Inclui descrições de TOTAL, IMS e System 2000.
- 1.11 I. R. Palmer. "Data Base Systems: A Practical Reference". Q. E. D. Information Sciences Inc., 141 Linden Street, Wellesley, Mass. 02181 (junho de 1975).
Contém referências sobre 20 sistemas implementados, bem como uma lista com 50 outros, além de alguns casos interessantes para estudo.
- 1.12 D. A. Jardine (ed.). *Data Base Management Systems: Proceedings of the SHARE Working Conference on Data Base Management Systems, Montreal, Canada, July 23-27, 1973*. North-Holland (1974).
Este livro inclui artigos sobre requisitos dos usuários, tendências futuras, experiências dos usuários com vários sistemas (IMS, TOTAL, IDS, DMS 1100, e EDMS), e declarações de planos e intenções de diversos fornecedores (Burroughs, Cincom, CDC, Honeywell, IBM, UNIVAC, e XDS).
- 1.13 W. C. McGee. "Generalized File Processing". *Annual Review in Automatic Programming*, Vol. 5 (eds., Halpern and Shaw). Elmsford, N.Y.: Pergamon Press (1969).
Este artigo foi o primeiro a introduzir o termo "esquema" (embora não exatamente com o significado da ANSI/SPARC). Um levantamento bastante lido com vários exemplos obtidos dos primeiros sistemas.
- 1.14 E. H. Sibley. "The Development of Data Base Technology". Guest Editor's Introduction to *ACM Comp. Surv.* 8, nº 1: Special Issue on Data Base Management Systems (março de 1976).
- 1.15 J. P. Fry and E. H. Sibley. "Evolution of Data Base Management Systems". *ACM Comp. Surv.* 8, nº 1 (março de 1976).
- 1.16 R. W. Engles. "A Tutorial on Data Base Organization". *Annual Review in Automatic Programming*, Vol. 7 (eds., Halpern and McGee). Elmsford, N. Y.: Pergamon Press (1974).
Uma boa introdução aos conceitos de bancos de dados. Os tópicos maiores incluídos são: teoria de dados operacionais, pesquisa sobre estruturas de armazenamento e técnicas de acesso, além de uma discussão sobre independência de dados.
- 1.17 ANSI/X3/SPARC Study Group on Data Base Management Systems. Artigo provisório. FDT (ACM SIGMOD bulletin) 7, nº 2 (1975).
- 1.18 D. C. Tsichritzis and A. Klug (eds.). "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems". *Information Systems* 3 (1978).
Estes dois documentos são os relatórios provisório e final do chamado ANSI/SPARC Study Group. O ANSI/X3/SPARC Study Group on Data Base Management Systems (para dar o título completo) foi estabelecido no final de 1972 pelo Standards Planning and Requirements Committee (SPARC) do ANSI/X3. ANSI/X3 é o American National Standards Committee on Computers and Information Processing. Os objetivos do Grupo de Estudo foram os de determinar as áreas, se existentes, de tecnologia de bancos de dados onde fosse apropriada uma atividade de padronização, e de produzir um conjunto de recomendações para ações em cada uma dessas áreas. Trabalhando sobre esses objetivos, o Grupo de Estudos verificou que os *interfaces* seriam os únicos aspectos do sistema de bancos de dados possivelmente passíveis de padronização, e assim definiram uma arquitetura genérica ou estrutura básica para um sistema de banco de dados e seus interfaces. O Relatório Final apresenta uma descrição detalhada dessa arquitetura e de alguns dos 42 interfaces identificados. O Relatório Provisório é um documento de

trabalho anterior, mas ainda assim interessante; ele fornece detalhes adicionais em algumas áreas.

- 1.19 D. A. Jardine (ed.). *The ANSI/SPARC DBMS Model: Proceedings of the Second SHARE Working Conference on Data Base Management Systems, Montreal, Canada, April 26-30, 1976*. North-Holland (1977).

Anais de uma conferência (artigos e discussões) dedicada às propostas da ANSI/SPARC documentadas no relatório provisório [1.17].

- 1.20 B. Yormark. "The ANSI/X3/SPARC/SG DBMS Architecture," Em [1.19]. Uma visão geral do relatório provisório [1.17].

- 1.21 B. Langefors. "Theoretical Analysis of Information Systems". Lund, Sweden (1966, 1973).

- 1.22 B. Sundgren. "Conceptual Foundations of the Infological Approach to Data Bases". *Proc. IFIP TC-2 Working Conference on Data Base Management Systems* (eds., Klimbie and Kofferman). North-Holland (1974).

Os trabalhos de Langefors [1.21] e Sundgren têm muitos pontos em comum com as propostas ANSI/SPARC, que eles anteciparam em vários anos. Até recentemente, entretanto, suas idéias não receberam muita aceitação geral, exceto na Escandinávia.

- 1.23 P. P. Uhrowczik. "Data Dictionary/Directories". *IBM Sys. J.* 12, nº 4 (1973).

Uma introdução aos conceitos básicos do sistema de dicionário de dados. É esboçada uma implementação usando bancos de dados IMS físicos e lógicos (veja a parte 3 deste livro).

- 1.24 Data Dictionary Systems Working Party of the British Computer Society. Relatório. Edição conjunta: Data Base (ACM SIGBDP newsletter) 9, nº 2; SIGMOD Record (ACM SIGMOD bulletin) 9, nº 4 (dezembro de 1977).

Uma excelente descrição do papel do dicionário de dados; inclui uma discussão breve porém interessante sobre o esquema conceitual.

- 1.25 M. R. Stonebraker. "A Functional View of Data Independence". *Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control*.

Este artigo representa um esforço para dar uma base precisa ao manuseio com problemas de independência de dados. Foram identificadas e rigorosamente definidas sete classes de transformações de um banco de dados armazenado de uma representação para outra. Os seguintes exemplos dão uma idéia dos tipos de transformações em cada uma das sete classes.

1. Relocalização física de arquivos armazenados
2. Conversão de valores de campos armazenados de um tipo de dado para outro
3. Substituição de um algoritmo de randomização por outro
4. Adição de índices
5. Duplicação de dados armazenados
6. Divisão de um registro armazenado em dois
7. Combinação de dois registros armazenados em um

É proposto que a medida do grau de independência de dados fornecida por um determinado sistema seja obtida considerando-se as transformações suportadas (de forma independente dos dados) em cada uma das sete classes. O artigo conclui com um rápido exame de três sistemas específicos à luz dessas idéias: RDMS, ISAM e IMS.

- 1.26 CODASYL Systems Committee. "Distributed Data Base Technology – An Interim Report". *Proc. NCC 47* (junho de 1978).

- 1.27 American National Standard Programming Language PL/I. ANSI Document X3.53-1976. American National Standards Institute, Inc. 1430 Broadway, New York, NY 10018.

2

Estruturas de Armazenamento

2.1 INTRODUÇÃO

O objetivo deste capítulo é o de fornecer uma introdução à forma como os dados podem ser organizados na memória secundária. Por "memória secundária" aqui queremos realmente significar os atuais dispositivos de acesso direto, tais como unidades de discos, tambores e assim por diante. Em outras palavras:

- Não estamos considerando meios puramente seqüenciais, como fitas, pois estas em geral impõem demasiadas restrições para que possam ser úteis em um sistema de banco de dados¹ (por exemplo, não é geralmente possível executar uma "atualização no local" em fita).
- Não estamos tentando predizer como serão os futuros meios. Note-se, entretanto, que se um novo dispositivo — digamos uma memória associativa de grande capacidade — vier a ser produzido, as empresas poderão se beneficiar de imediato, desde que haja independência de dados.

Outro lembrete introdutório: Neste capítulo nós não estaremos preocupados com sistemas específicos. As estruturas de armazenamento existentes nos sistemas individuais serão consideradas com alguma abrangência em capítulos posteriores.

Este capítulo, então, está voltado para o nível interno do sistema (e também, até certo ponto, com o que se encontra abaixo desse nível — lembre-se de que o nível interno ainda está ligeiramente acima do nível físico de armazenamento). Como explicado no Capítulo 1, as operações do usuário são expressas (via a DML) em termos de registros externos, e têm que ser convertidas pelo DBMS nas correspondentes operações sobre registros internos ou armazenados.

¹ As fitas são usadas nos sistemas de bancos de dados como meios para operações de controle de uso e descarga. Entretanto, não se usam normalmente fitas para o armazenamento do banco de dados operacional propriamente dito.

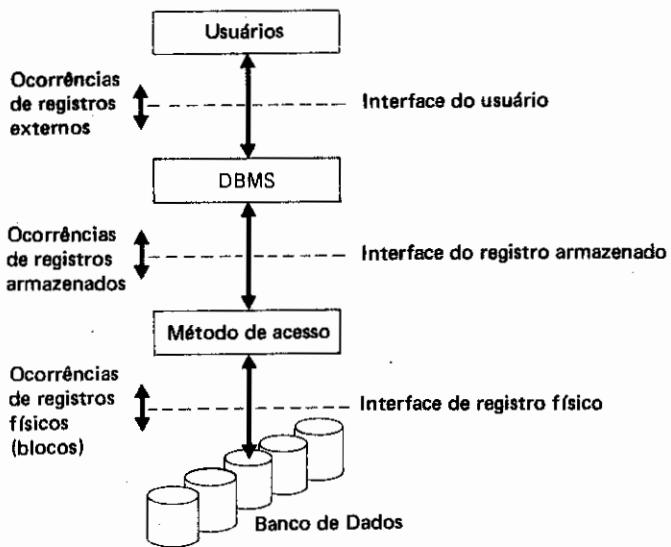


Fig. 2.1 Interface do registro armazenado

Essas últimas operações ainda têm que ser convertidas em operações ao nível do *hardware* em uso, isto é, a operações sobre registros ou blocos físicos. O componente responsável pela conversão interna/física é chamado de *método de acesso* (Fig. 2.1). O método de acesso consiste de um conjunto de rotinas cuja função é ocultar do DBMS todos os detalhes dependentes de dispositivos e apresentar um *interface²* do registro armazenado ao DBMS. Portanto, o interface do registro armazenado corresponde ao nível interno, da mesma forma que o interface do usuário corresponde ao nível externo. [O interface do registro físico (veja Fig. 2.1) corresponde ao nível do *hardware* usado].

[O termo ANSI/SPARC para o método de acesso é “subsistema de armazenamento”. Como um aparte, podemos notar que o método de acesso pode ser visto como uma extensão lógica do DBMS, e sem dúvida, em alguns sistemas, os dois são acondicionados juntos. Em outros casos, o DBMS conta com o sistema operacional básico para prover a função do método de acesso. Para firmar nossas idéias vamos usar esta última estrutura.]

O interface do registro armazenado permite que o DBMS veja a estrutura de armazenamento como uma coleção de arquivos armazenados, cada um consistindo de todas as ocorrências de um tipo de registro armazenado (veja o Capítulo 1). Especificamente, o DBMS sabe (a) que arquivos armazenados existem, e, para cada um, (b) a estrutura do correspondente registro armazenado, (c) os campos armazenados, se existentes, pelos

² Muitos métodos de acesso atualmente existentes fazem bastante mais. Portanto, muitas das funções que a seguir são atribuídas ao DBMS podem, na prática, ser executadas pelo método de acesso; por exemplo, o método de acesso pode prover seus próprios índices secundários. Entretanto, preferimos escolher um método de acesso bem mais elementar como base mais adequada para a descrição das estruturas de armazenamento possíveis. A existência de métodos de acesso mais complexos não invalida em nada as discussões subsequentes.

quais está feita a seqüenciação, e (d) os campos armazenados, se existentes, que podem ser usados como argumentos de pesquisa para acesso direto. Toda esta informação deve ser especificada como parte da definição da estrutura de armazenamento. Note que os itens (c) e (d) significam dizer-se que o DBMS sabe que instruções do método de acesso ele pode emitir sobre o arquivo armazenado. Note, também, que a unidade que cruza o interface do registro armazenado é uma ocorrência de registro armazenado.

O DBMS *não* sabe (a) nada sobre registros físicos (blocos); (b) como os campos armazenados estão associados para formarem os registros armazenados (muito embora na prática isto quase invariavelmente ocorra por adjacência física); (c) como foi feito o seqüenciamento (por exemplo, pode ser por meio de contigüidade física, um índice, ou uma cadeia de indicadores de localização); ou (d) como é executado o acesso direto (por exemplo, pode ser via um índice, pesquisa seqüencial, ou endereçamento randômico). Esta informação é especificada para o método de acesso, não para o DBMS.

Como uma ilustração, vamos imaginar que a estrutura de armazenamento inclui um arquivo armazenado de peças (**PART**), no qual cada ocorrência de registro armazenado consiste de um número de peça (**P #**), um nome de peça (**PNAME**), uma cor (**COLOR**), e um peso (**WEIGHT**). Parte da definição da estrutura de armazenamento poderia então ser como se segue:

```
STORED FILE PART_FILE
    PART LENGTH = . . . , SEQUENCE = ASCENDING(P#)
        P#      TYPE = . . . , INDEX = P#_INDEX
        PNAME   TYPE = . . .
        COLOR   TYPE = . . .
        WEIGHT  TYPE = . . .
```

Esta definição pretende transmitir a seguinte informação.

- O arquivo armazenado de peças (**PART**) existe
- O registro armazenado correspondente possui uma estrutura específica
- As ocorrências de registros armazenados estão seqüenciadas de acordo com os valores crescentes **P#** (note que o mecanismo de seqüenciamento *não* está especificado).
- O campo armazenado **P#** está indexado, de forma que se pode fazer um acesso direto fornecendo-se um valor para este campo. (Naturalmente, é possível que se tenha que especificar muitos outros detalhes da estrutura de armazenamento também.)

Vamos fazer outra suposição: Quando uma nova ocorrência de registro armazenado é inicialmente criada e colocada dentro do banco de dados, o método de acesso é responsável por designar a ele um único *endereço de registro armazenado* (SRA). Este valor distingue esta ocorrência de registro armazenado de todas as outras no banco de dados. Ele pode, por exemplo, ser simplesmente o endereço físico da ocorrência dentro do volume de armazenamento (juntamente com a identificação do volume), ou, alternativamente, uma identificação do arquivo armazenado apropriado juntamente com um deslocamento dentro daquele arquivo (considerando-se o arquivo como uma seqüência de bytes). O SRA de uma determinada ocorrência é informado pelo método de acesso ao DBMS quan-

do a ocorrência é inicialmente criada, e pode ser usado pelo DBMS para acesso direto subsequente àquela ocorrência. Estamos supondo aqui que o SRA de uma determinada ocorrência não muda até que a ocorrência seja fisicamente movimentada como parte de uma reorganização do banco de dados (se então isto ocorrer).

O conceito de SRA permite que o DBMS construa o seu próprio mecanismo de acesso (índices, cadeias de indicadores de localização etc.), adicionalmente aos que são mantidos pelo método de acesso. Isto encontra-se ilustrado por meio de exemplos na Seção 2.2.

2.2 REPRESENTAÇÕES POSSÍVEIS PARA ALGUNS DADOS DE TESTE

Nesta seção vamos tomar uma coleção simples de dados de exemplo e considerar as várias maneiras de como eles poderiam estar representados no armazenamento (ao nível do interface do registro armazenado). Os dados de exemplo estão mostrados na Fig. 2.2. Eles consistem de informações sobre cinco fornecedores; para cada fornecedor desejamos registrar um número de fornecedor (S#), um nome de fornecedor (SNAME), um valor de estado (STATUS), e uma localização (CITY). Estamos supondo que cada fornecedor possui um único número de fornecedor, e também que cada fornecedor tem um nome, um valor de estado e uma localização.

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Fig. 2.2 Dados de exemplo

No que se segue, estaremos supondo, a menos que seja explicitamente indicado de outra forma, que cada arquivo armazenado está seqüenciado pelo método de acesso por sua *chave primária*. [Este termo será rigorosamente definido no Capítulo 4; no momento vamos supor que ele está bem entendido; basicamente ele significa um campo (combinação) cujos valores identificam univocamente os registros do arquivo.] O mecanismo pelo qual este seqüenciamento é feito não será mostrado.

A primeira (e mais simples) representação consiste de um arquivo armazenado singelo contendo cinco ocorrências de registros armazenados, um para cada fornecedor. A figura 2.2 pode ser considerada como uma ilustração disto. Esta representação apresenta a vantagem da simplicidade, mas seria provavelmente inadequada na prática. Suponhamos, por exemplo, que tivéssemos 10.000 fornecedores ao invés de apenas cinco, mas que eles estivessem localizados em dez cidades diferentes. Se partirmos do princípio de que a quantidade de memória necessária para um *indicador de localização* é menor do que a requerida para um nome de cidade, a representação ilustrada na Fig. 2.3 irá certamente economizar algum espaço de armazenamento nessa situação.

Arquivo de FORNECEDORES

S#	SNAME	STATUS	Indicador da localização em CITY	Arquivo
S1	Smith	20	●	CITY
S2	Jones	10	●	Athens
S3	Blake	30	●	London
S4	Clark	20	●	Paris
S5	Adams	30	●	

Fig. 2.3 Retirando os valores CITY.

Aqui nós temos dois arquivos armazenados; um arquivo de fornecedores e um arquivo de cidades, com indicadores de localização apontando do primeiro para o segundo. Estes indicadores de localização são SRAs. A única vantagem desta representação (comparada com a anterior) é a economia de espaço. Por exemplo, uma solicitação para encontrar todas as propriedades de um determinado fornecedor irá requerer pelo menos mais um acesso do que antes; uma solicitação para encontrar todos os fornecedores de uma determinada cidade envolveria diversos acessos adicionais. Note, incidentalmente, que é o DBMS, não o método de acesso, que mantém os indicadores de localização; o método de acesso não está a par da conexão entre os dois arquivos. (A conexão tem que estar expressa na definição do mapeamento do esquema conceitual para a memória.)

Se a consulta “encontre todos os fornecedores de uma determinada cidade” for importante, o DBA pode escolher a representação alternativa mostrada na Fig. 2.4. Aqui nós temos novamente dois arquivos, um arquivo de fornecedores e um arquivo de cidades, mas desta vez os indicadores de localização partem do último para o primeiro (cada ocorrência de registro armazenado de cidade contém indicadores das localizações de todas as ocorrências correspondentes de registros armazenados de fornecedores).

É óbvio que esta representação é melhor do que a anterior para consultas que solicitem todos os fornecedores de uma determinada cidade, porém é pior para consultas que solicitem todos os atributos de um dado fornecedor. Os requisitos de memória são basicamente os mesmos.

Arquivo

CITY	Indicadores de localização de FORNECEDORES		Arquivo de FORNECEDORES
	S#	SNAME	
Athens	S1	Smith	20
London	S2	Jones	10
Paris	S3	Blake	30
	S4	Clark	20
	S5	Adams	30

Fig. 2.4 Indexação em CITY.

O arquivo de cidade na Fig. 2.4 serve como um *índice* para o arquivo de fornecedores (equivalentemente, o arquivo de fornecedores está *indexado pelo* arquivo de cidade). O objetivo de um índice é o de prover um *caminho de acesso* ao arquivo que ele está indexando — isto é, uma maneira de se alcançar os registros no arquivo indexado. Um determinado arquivo pode ter vários caminhos de acesso associados. Um caminho que está sempre disponível é o seqüencial simples — é sempre possível fazer-se uma pesquisa exaustiva ao arquivo registro por registro, de acordo com o seqüenciamento básico (normalmente a seqüência da chave primária) com o suporte fornecido pelo método de acesso. Mais tarde nesta seção vamos discutir outros tipos de caminhos de acesso.

Retornando ao tópico de indexação: Um índice, portanto, é um arquivo no qual cada entrada (registro) consiste de um valor de dado juntamente com um ou mais indicadores de localização. O conteúdo de dado é um valor para algum campo do arquivo indexado (o *campo indexado*), e os indicadores de localização identificam registros no arquivo indexado que possuem aquele valor para aquele campo. Um índice pode ser usado de duas maneiras. Primeiro, ele pode ser usado para acesso *seqüencial* ao arquivo indexado — acesso seqüencial, isto é, de acordo com os valores do campo indexado. (Em outras palavras, ele determina uma ordenação do arquivo indexado.) Segundo, ele pode também ser usado para acesso *direto* aos registros individuais do arquivo indexado na base de um dado valor para aquele mesmo campo.

Em geral, portanto, a vantagem da indexação é que (usualmente) ela acelera a recuperação da informação. A desvantagem é a de que ela pode tornar mais lenta a atualização. Por exemplo, suponha que o fornecedor S2 se muda de Paris para Londres, e imagine o que teria que ser feito para refletir esta mudança na Fig. 2.4, em comparação com o que teria que ser feito na Fig. 2.3.

Note, incidentalmente, que o arquivo de cidade mostrado na Fig. 2.4 é um índice controlado pelo DBMS, e não pelo método de acesso. Ele é de fato um índice *denso, secundário*. “Denso” significa que ele contém uma entrada para cada ocorrência de registro armazenado no arquivo indexado; isto quer dizer que o arquivo indexado não precisa conter o campo indexado — no nosso exemplo, o arquivo de fornecedores não mais contém o campo *cidade*. (Nós vamos abordar a indexação *não-densa* na seção 2.3.) “Secundário” significa que este é um índice em um campo outro que não a chave primária.

A Fig. 2.5 mostra uma representação de dados que combina as duas anteriores, possuindo portanto as vantagens de cada uma. Naturalmente esta também apresenta a desvantagem para atualização mencionada anteriormente, além de requerer um pouco mais de memória.

Outra desvantagem dos índices *secundários* em geral é que cada ocorrência de registro armazenado no índice tem que conter um número imprevisível de indicadores de localização (porque usualmente o campo indexado não terá um conteúdo distinto para cada ocorrência no arquivo indexado). Este fato complica o trabalho do DBMS na efetivação de mudanças no banco de dados. Uma alternativa à representação prévia que evita este problema está ilustrada na Fig. 2.6.

Nesta representação cada ocorrência de registro armazenado (fornecedor ou cidade) contém somente um indicador de localização. Cada cidade indica o primeiro fornecedor naquela cidade. Aquele fornecedor então indica o segundo fornecedor na mesma cidade, que indica o terceiro, e assim sucessivamente até o último, que indica de volta a cidade. Portanto, para cada cidade nós temos uma *cadeia* de todos os fornecedores naquela cidade (outro exemplo de caminho de acesso). A vantagem desta representação é a de que se torna fácil a execução de modificações. A desvantagem é que, para uma dada cidade,

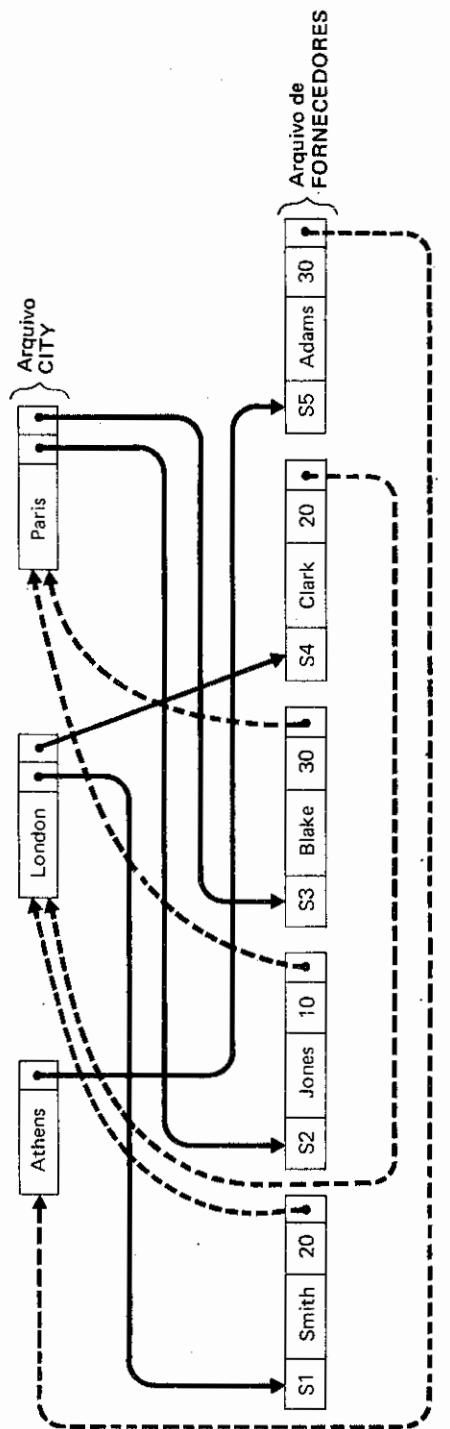


Fig. 2.5 Combinagem das duas representações anteriores.

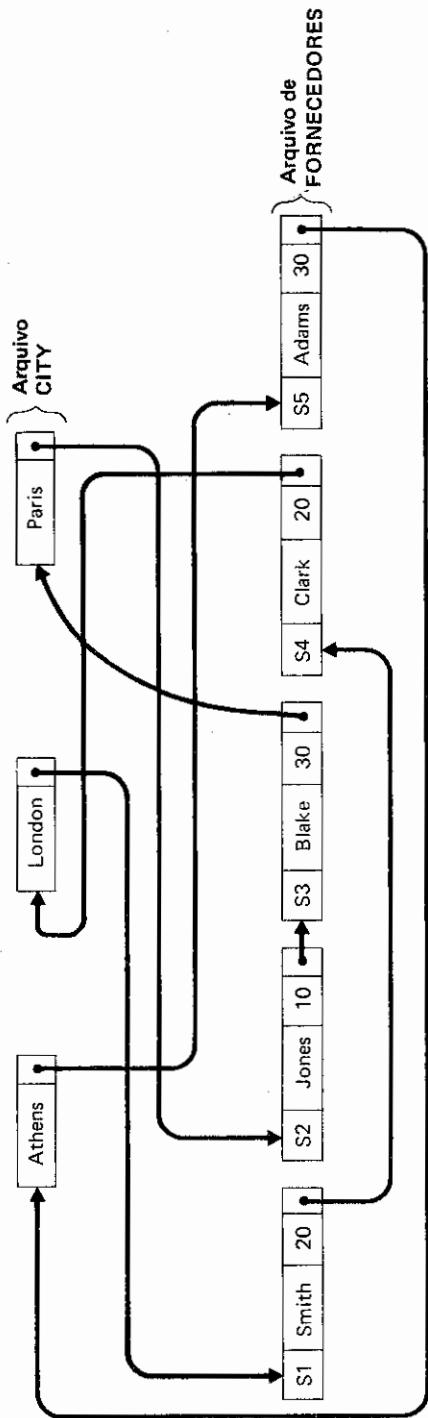


Fig. 2.6 Uso de cadeias de indicadores de localização.

o único caminho de acesso ao enésimo fornecedor é seguindo a cadeia passando pelo primeiro, segundo, . . . , ($n - 1$) fornecedores. Se cada acesso envolver uma operação de busca (*seek*), o tempo absorvido para se ter acesso ao enésimo fornecedor pode ser considerável.

Como uma extensão da representação anterior, nós poderíamos tornar as cadeias com dupla direção (de tal forma que cada ocorrência de registro armazenado contenha exatamente *dois* indicadores de localização). Esta representação poderia ser escolhida se a eliminação de fornecedores for uma operação comum, por exemplo, pois ela simplifica o processo de ajustamento dos indicadores de localização necessário em tal operação. Outra extensão poderia ser a inclusão de um indicador de localização em cada fornecedor apontando diretamente para a cidade correspondente (como nós fizemos na Fig. 2.3), para reduzir o percurso na cadeia requerido por certos tipos de consulta.

A representação mostrada na Fig. 2.6 (usando cadeias de indicadores de localização) é um exemplo simples da *organização em listas múltiplas*. Na Fig. 2.6 nós encadeamos todos os fornecedores na mesma cidade; em outras palavras, para cada cidade nós fizemos uma lista dos fornecedores correspondentes. Exatamente da mesma maneira (por meio de indicadores de localização adicionais) poderíamos ter preparado listas de fornecedores para cada valor distinto de estado (*status*), por exemplo. Sugiro que o leitor tente projetar esta representação para os dados de teste. Em geral, uma organização em listas múltiplas pode conter qualquer número dessas listas.

Retornando à indexação secundária: Da mesma forma como é possível ter-se qualquer quantidade de listas em uma organização de listas múltiplas, é também possível ter-se qualquer quantidade de índices secundários em uma organização indexada. No caso extremo, nós podemos ter a situação ilustrada na Fig. 2.7; um índice em cada campo secundário, ou *organização invertida*. (O símbolo ↑ é usado para significar “indica a localização de”.)

Entretanto, embora a organização invertida se desempenhe bem para solicitações de todos os fornecedores com uma determinada propriedade (digamos, um estado de 20), exigirá maior tempo para responder a uma consulta sobre todas as propriedades de determinado fornecedor. Na prática, entretanto, procuramos um acordo, montando uma organização *comum* (como a mostrada na Fig. 2.2) juntamente com índices secundários em certos campos selecionados. (Note que isto traz redundância de memória para os valores dos campos indexados.) Esta é uma das estruturas de armazenamento mais comuns em uso corrente.

Índice SNAME		Índice STATUS		Índice CITY		Arquivo de FORNECEDORES
SNAME	Indicadores de localização	STATUS	Indicadores de localização	CITY	Indicadores de localização	S#
Smith	↑ S1	10	↑ S2	Athens	↑ S5	S1
Jones	↑ S2	20	↑ S1, ↑ S4	London	↑ S1, ↑ S4	S2
Blake	↑ S3	30	↑ S3, ↑ S5	Paris	↑ S2, ↑ S3	S3
Clark	↑ S4					S4
Adams	↑ S5					S5

Fig. 2.7 Organização invertida.

Outra representação que poderia ser mencionada é a organização *hierárquica*, ilustrada na Fig. 2.8. Aqui nós temos um arquivo armazenado contendo três ocorrências (hierárquicas) de registros armazenados, uma para cada cidade. Parte de cada ocorrência de registro armazenado consiste de uma lista de tamanho variável contendo entradas de fornecedores, uma para cada fornecedor naquela cidade, e cada entrada de fornecedor contém um número de fornecedor, nome e estado. (Uma lista de tamanho variável contida em um registro como este é algumas vezes chamada de *grupo repetitivo*.)

Como em muitos exemplos anteriores, nós sepamos os valores de CITY, mas desta vez escolhemos, para representar a associação entre uma cidade e seus fornecedores, colocar a cidade e os fornecedores como parte de uma ocorrência de registro armazenado (ao invés de se usar indicadores de localização, como na Fig. 2.4, por exemplo). Incidentalmente, um índice secundário como o da Fig. 2.4 é, na realidade, um arquivo hierárquico.

A última representação que vamos considerar é uma organização de *endereçamento randômico*. Este tipo de endereçamento é outro exemplo de um caminho de acesso. A idéia básica do endereçamento randômico é a de que cada ocorrência de registro armazenado está colocada no banco de dados em uma localização cujo endereço (SRA) pode ser calculado como alguma função (*a função de randomização*) de um valor que apareça naquela ocorrência — normalmente o valor da chave primária. Portanto, para armazenar inicialmente a ocorrência, o DBMS calcula o SRA e instrui o método de acesso para colocar a ocorrência naquela posição; e para recuperar subsequentemente a ocorrência, o DBMS executa o mesmo cálculo que antes e solicita que o método de acesso vá buscar a ocorrência na posição calculada. A vantagem desta organização é que ela fornece um acesso direto muito rápido na base dos valores do campo usado para o cálculo.

Como um exemplo de endereçamento randômico, vamos supor que os valores de S# são S100, S200, S300, S400, S500 (ao invés de S1, S2, S3, S4, S5), e vamos considerar como função de *randomização*:

$$\text{SRA} = \text{resto após a divisão do valor de S\# (sua parte numérica) por 13.}$$

(um exemplo simples de uma classe bastante comum de função de randomização — “divisão/resto”).³ Os SRAs para os cinco fornecedores são então 9, 5, 1, 10, 6, respectivamente, dando a representação mostrada na Fig. 2.9 (onde nós estamos supondo que esses valores de SRA representam simplesmente posições dos registros dentro do arquivo armazenado).

Além de mostrar como esse tipo de endereçamento trabalha, o exemplo também mostra por que é necessária a função de randomização. Seria teoricamente possível usar

Athens			London			Paris		
S5	Adams	30	S1	Smith	20	S2	Jones	10
S4	Clark	20	S3	Blake	30	S3	Blake	30

Fig. 2.8 Organização hierárquica.

3

Por motivos que fogem ao escopo deste livro, normalmente, escolhe-se para divisor um número primo.

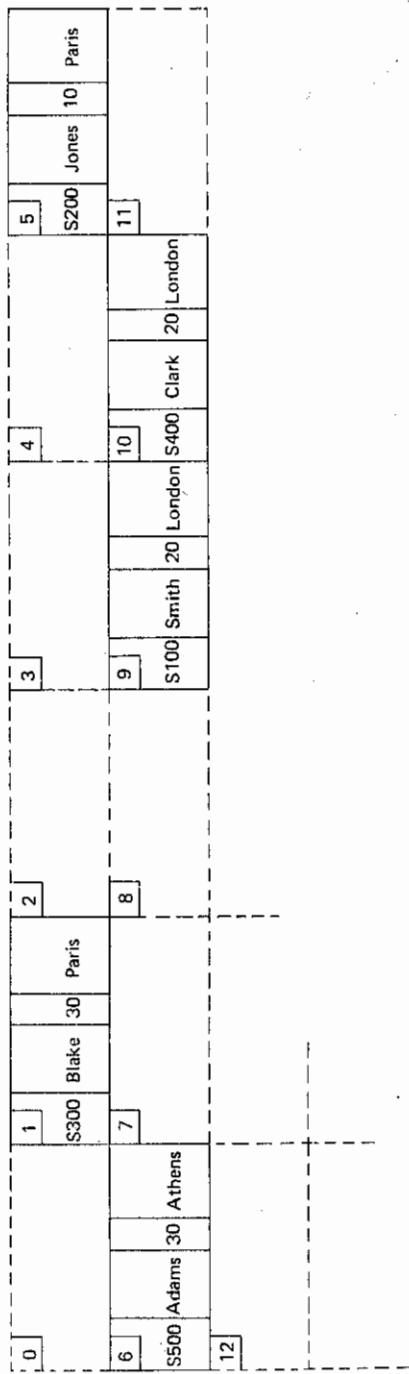


Fig. 2.9 Uma organização de endereçamento randômico.

uma função de randomização “identidade”, isto é, usar o valor (numérico) da chave primária de uma determinada ocorrência diretamente como o SRA para aquela ocorrência. Entretanto, isto é normalmente inadequado na prática, pois a faixa de valores de chaves primárias é geralmente muito mais abrangente do que a faixa de SRAs disponíveis. Por exemplo, suponhamos que os números de fornecedores possuem de fato três dígitos, como especificamos acima. Isto permite um número máximo teórico de 1.000 fornecedores diferentes, enquanto que na prática podemos estar usando no máximo dez. Para evitar um desperdício colossal de espaço de armazenamento, nós idealmente precisamos de uma função de randomização que reduza qualquer valor na faixa 0–999 a outro na faixa de 0–9. Com o objetivo de permitir futuro crescimento, é usual que se estenda a faixa desejada em uns 20 por cento mais ou menos; portanto, no exemplo, nós realmente escolhemos uma função que gera valores na faixa de 0–12 ao invés de 0–9.

O exemplo também ilustra uma das desvantagens do endereçamento randômico: a seqüência de ocorrências de registros armazenados dentro do arquivo armazenado quase certamente não seguirá a seqüência das chaves primárias (além de poderem existir intervalos de tamanho arbitrário entre ocorrências consecutivas). De fato, um arquivo armazenado segundo essa organização é usualmente (não invariavelmente) tido como não possuindo qualquer sequenciação específica.

Outra desvantagem desse tipo de endereçamento é a possibilidade de *colisões* — isto é, duas ocorrências distintas de registros armazenados cujas chaves endereçam para o mesmo SRA. Por exemplo, suponhamos que os dados de teste incluem também um fornecedor cujo valor de S# é 1400. Este fornecedor irá colidir (no SRA 9) com o fornecedor S100, se usada a mesma função que antes. A implicação é que a função de randomização tem que ser mais elaborada para manipular este tipo de situação. Uma possibilidade é tornar o SRA (obtido como acima) o ponto inicial de uma varredura seqüencial. Portanto, para inserirmos o fornecedor S1400 (supondo que já existam os fornecedores S100 – S500), nós iremos para o SRA 9 e pesquisaremos para frente a partir dessa posição, em busca da primeira localização livre. O novo fornecedor será armazenado no SRA 11. Para subsequentemente recuperar-se o fornecedor S1400, seguiremos um processo semelhante.

Nosso levantamento de algumas estruturas de armazenamento permitidas pelo interface do registro armazenado está agora completo. Como conclusão devemos enfatizar que não existe a “melhor” estrutura de armazenamento. O que é “melhor” depende do que é importante para a empresa. É responsabilidade do DBA balancear uma grande quantidade de requisitos conflitantes ao escolher a estrutura de armazenamento. Entre as considerações que têm que ser feitas incluem-se: desempenho na recuperação, dificuldade de execução de mudanças, quantidade de espaço de armazenamento disponível, facilidade para reorganizar o banco de dados e a freqüência desejada para tais reorganizações, problemas de recuperação, e assim por diante.

2.3 INTERFACE DO REGISTRO FÍSICO: TÉCNICAS DE INDEXAÇÃO

O interface do registro físico é o que se encontra entre o método de acesso e o banco de dados físico. A unidade que cruza este interface é uma ocorrência de registro físico (um bloco). A diferença realmente significativa entre este interface e o interface do registro armazenado discutido na Seção 2.2 é que aqui o conceito de contigüidade física torna-se significativo. A contigüidade física fornece um meio importante para se representar a seqüência de ocorrências de registros armazenados dentro do arquivo armazenado, fazendo

uso de (a) a seqüência física das ocorrências de registros armazenados dentro de um bloco, e (b) a seqüência física de blocos no meio. Isto traz duas implicações na área de projetos de índices.

Primeira, é possível construir-se índices *não densos*. A idéia aqui é a de que o arquivo que está sendo indexado seja dividido em grupos, com diversas ocorrências de registros armazenados em cada grupo, de tal forma que prevaleçam as seguintes condições.

- Considerados dois grupos quaisquer, todas as ocorrências de registros armazenados em um precedem todas as do outro (em relação ao seqüenciamento estabelecido para o arquivo).
- Dentro de um grupo qualquer, a seqüência do arquivo é representada pela contigüidade física.

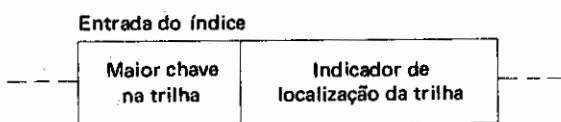
(Na prática, um grupo pode ser um bloco, uma trilha, ou qualquer outra unidade conveniente de espaço de armazenamento). O índice então contém uma entrada por grupo, fornecendo (tipicamente) o maior valor de campo indexado existente no grupo, e um indicador de localização do ponto inicial do grupo. A seqüência de grupos fica representada pela seqüência de entradas no índice.

O termo “não denso” refere-se ao fato do índice *não* conter uma entrada para cada ocorrência de registro armazenado no arquivo indexado. Por isso as ocorrências de registros armazenados *têm* que conter o campo indexado (em contraste com a situação de índice denso – Fig. 2.4). A Fig. 2.10 ilustra a situação em que o grupo é uma trilha.

O segundo aspecto que surge da disponibilidade de seqüenciamento físico é a possibilidade de se construir índices em *múltiplos níveis* (estruturas em árvore). A razão para se colocar um índice no ponto inicial é a de se remover a necessidade de varredura seqüencial no arquivo indexado. Entretanto, ainda é necessária uma varredura seqüencial no índice. Se o índice tornar-se muito extenso, este fato em si pode causar uma perda significativa de desempenho. A solução é estabelecer-se um índice do índice. Para um exemplo comum, veja a Fig. 2.11.

Aqui o arquivo indexado está dividido em grupos de uma trilha cada. O *índice de trilha* contém uma entrada para cada uma dessas trilhas (como na Fig. 2.10). O índice de trilha, por sua vez, está dividido em grupos, cada um dos quais consistindo das entradas para todas as trilhas de um cilindro do arquivo indexado, e um *índice de cilindro* contendo uma entrada para cada um dos grupos de índices de trilhas.

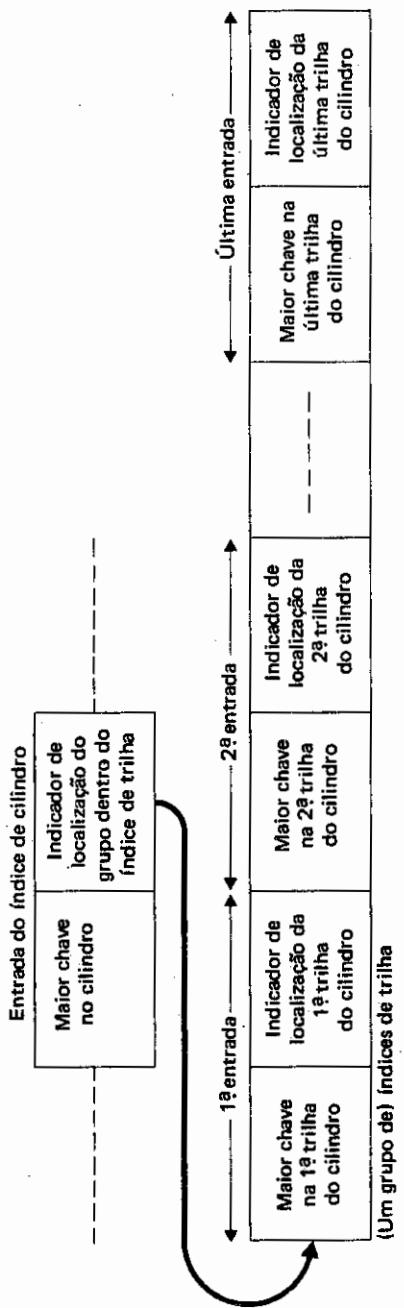
(Cada grupo dentro do índice de trilha é normalmente gravado no início do cilindro apropriado do arquivo indexado, para reduzir a atividade de procura.)



PROCESSO DE ACESSO:

1. Pesquisar no índice chave \geq chave requerida.
2. Ir para a trilha indicada.
3. Executar varredura física seqüencial na trilha.

Fig. 2.10 Exemplo de indexação não densa



PROCESSO DE ACESSO:

1. Pesquisar no índice de cilindro chave > chave requerida.
2. Ir para o grupo indicado dentro do índice de trilha.
3. Pesquisar o grupo no índice de trilha para chave ≥ chave requerida.
4. Ir para a trilha indicada.
5. Executar varredura física sequencial na trilha.

Fig. 2.11 Exemplo de indexação em múltiplos níveis

Em geral, um índice em múltiplos níveis pode conter qualquer quantidade de níveis, cada um dos quais atuando como um índice não denso para o nível inferior. (Ele tem que ser não denso; caso contrário nada será obtido.) Analisando a técnica pela sua conclusão lógica, o nível mais alto deveria conter uma única entrada; na prática, entretanto, o nível mais alto consistindo de um único bloco (contendo muitas entradas) é o máximo que se pode exigir.

Árvores-B

Uma forma particular de índice em múltiplos níveis, ou estrutura em árvore, que se tornou extremamente popular nos últimos anos, é a *árvore-B*. As árvores-B foram descritas em um artigo de Bayer e McCreight em 1972 [2.16]. Desde aquela época foram propostas numerosas variações sobre a idéia básica por Bayer e outros investigadores. A variação apresentada por Knuth [2.1] é uma das técnicas mais comuns encontrada em sistemas modernos: em particular, a estrutura de índices do Virtual Storage Access Method, VSAM, da IBM, é similar à estrutura de Knuth [2.17]. (Entretanto, a versão VSAM foi inventada independentemente e inclui dispositivos próprios, tais como o uso de técnicas de compressão [a ser visto mais tarde]. De fato, um precursor da estrutura VSAM foi descrito já em 1969 por H. K. Chang [2.21].)

Na variante de Knuth, o índice consiste de duas partes: o *conjunto em seqüência* e o *conjunto índice* (terminologia do VSAM). O *conjunto em seqüência* consiste de um índice denso de um nível indicando os dados reais; as entradas no índice são blocadas, e os blocos são encadeados uns aos outros, de tal forma que a ordenação do arquivo representada pelo índice é obtida tomando-se as entradas (na ordem física) no primeiro bloco da cadeia, seguidas pelas entradas (na ordem física) no segundo bloco da cadeia, e assim por diante. Portanto, o conjunto em seqüência possibilita um acesso seqüencial rápido aos dados. O *conjunto índice*, por seu turno, possibilita acesso direto rápido ao conjunto em seqüência (e portanto também aos dados). O conjunto índice é na realidade um índice de estrutura em árvore indicando o conjunto em seqüência; de fato, o conjunto índice é a verdadeira árvore-B, estritamente falando. (A combinação do conjunto índice com o conjunto em seqüência é algumas vezes chamada de árvore-B^{*}.) Um exemplo simples está mostrado na Fig. 2.12.

Explicaremos a Fig. 2.12 a seguir. Os valores 6, 8, 12, ..., 97, 99 são valores do campo indexado, digamos F. Consideremos o nó mais alto, que consiste de duas entradas de valores (50 e 82) e três indicadores de localização. Registros de dados em que F é menor do que ou igual a 50 podem ser encontrados (eventualmente) seguindo o indicador de localização esquerdo a partir deste nó; de forma semelhante, registros nos quais F seja maior do que 50 e menor do que ou igual a 82 podem ser encontrados seguindo o indicador de localização central, e registros nos quais F seja maior do que 82 podem ser encontrados seguindo o indicador de localização direito. Os outros nós do conjunto índice são interpretados de forma semelhante; note, em particular, que (por exemplo) seguindo o indicador de localização direito a partir do primeiro nó do segundo nível nós chegamos a todos os registros cujo F é maior do que 32 e menor do que ou igual a 50 (pelo fato de termos seguido o indicador de localização esquerdo a partir do nó mais alto).

A árvore-B (conjunto índice) mostrado na Fig. 2.12 é, entretanto, algo irreal, pelas duas seguintes razões: Primeiro, normalmente os nós de uma árvore-B não contêm todos o mesmo número de entradas de valores; segundo, eles normalmente contêm uma certa disponibilidade de espaço vazio. Em geral, uma árvore-B de ordem n contém pelo menos

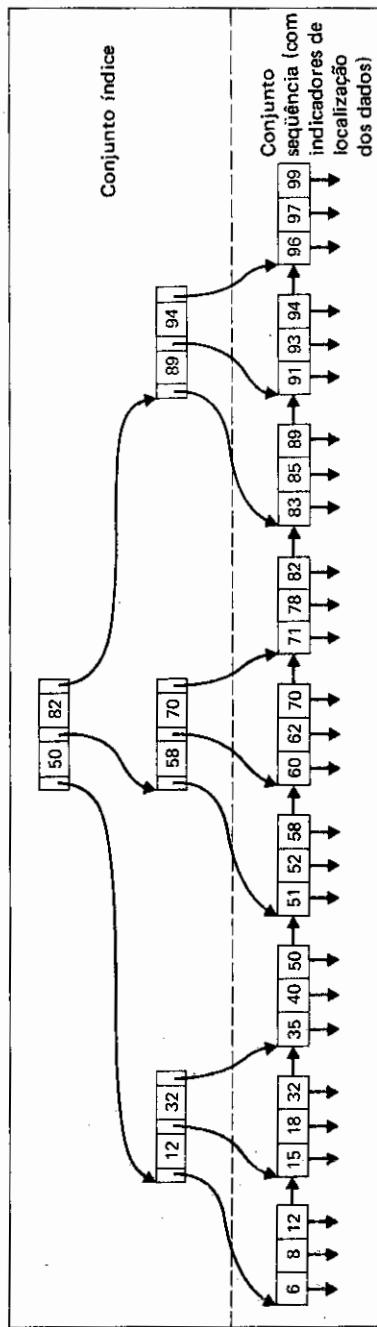


Fig. 2.12 Parte de uma árvore-B simples (Variante de Knuth).

n mas não mais do que $2n$ entradas de valores em um determinado nó (e, se ele tiver k entradas de valores, terá também $k + 1$ indicadores de localização). Nenhum valor de dado aparece na árvore mais de uma vez. Damos abaixo o algoritmo para pesquisar um determinado valor V na estrutura da Fig. 2.12; o algoritmo para a árvore-B genérica é uma simples generalização.

```
set N to the top node;
repeat until N is a sequence-set node;
    let X, Y be the data values in node N ( $X < Y$ );
    if  $V \leq X$ 
        then set N to the left lower node of N;
    if  $X < V \leq Y$ 
        then set N to middle lower node of N;
    if  $V > Y$ 
        then set N to right lower node of N;
end;
if V occurs in node N then exit (found);
if V does not occur in node N then exit (not found);
```

Em geral, um problema com as estruturas em árvore é que inserções e remoções podem tornar a árvore *desbalanceada*. Uma árvore está desbalanceada se os nós terminais não tiverem todos o mesmo nível – isto é, diferentes nós terminais encontram-se em “profundidades” diferentes abaixo do nó mais alto. Uma vez que a pesquisa às árvores envolve um acesso à memória secundária para cada nó verificado, os tempos de pesquisa podem tornar-se imprevisíveis em uma árvore desbalanceada. A vantagem notável das árvores-B é que o algoritmo de inserção/remoção da árvore-B garante que esta permanecerá balanceada. Vamos considerar brevemente a inserção de um novo valor, digamos V , em uma árvore-B de ordem n . (O algoritmo descrito manipula somente o conjunto índice, pois, como explicado anteriormente, o conjunto índice é que é a árvore-B propriamente. Basta uma extensão trivial para lidar também com o conjunto em seqüência.)

Primeiramente, o algoritmo é executado para localizar, não o conjunto em seqüência, mas o nó (digamos N) no menor nível do conjunto índice ao qual V pertence logicamente. Se N tiver espaço disponível, V é inserido em N e o processo termina. Senão, o nó N (que tem que conter $2n$ valores) será *dividido* em dois nós N_1 e N_2 ; os n menores valores do conjunto que constitui os $2n$ valores originais e mais o valor V serão colocados no nó esquerdo N_1 , e os n maiores valores do conjunto serão colocados no nó direito N_2 , sendo o valor intermediário, digamos W , promovido ao nó “pai” de N , digamos P , para servir como um valor de separação entre os nós N_1 e N_2 . (Futuras pesquisas de um valor V' , ao encontrarem o nó P , serão encaminhadas ao nó N_1 se $V' \leq W$ e ao nó N_2 se $V' > W$.) É feita então uma tentativa para inserir W em P , e o processo se repete. No pior caso, a divisão percorrerá todo o caminho até o topo da árvore, que crescerá um nível em altura (mas ainda assim permanecendo balanceada).

O algoritmo para remoção é essencialmente o inverso do procedimento que acabamos de descrever.

2.4 TÉCNICAS GERAIS DE INDEXAÇÃO

Na seção anterior apresentamos duas técnicas, indexação não densa e indexação em múltiplos níveis, que podem ser aplicadas ao interface do registro físico. Em tese, essas téc-

nicas poderiam ser aplicadas ao interface do registro armazenado também, mas esta aplicação não encontraria respaldo prático, pois tais técnicas dependem largamente da continuidade física. Em contraste, as seguintes técnicas podem ser usadas para índices em qualquer nível:

- Indexação por combinações de campos
- Seletividade no índice
- Técnicas de compressão
- Representação por indicador de localização simbólico

Indexação por combinações de campos

É possível construir-se um índice na base da combinação dos valores de dois ou mais campos. Por exemplo, a Fig. 2.13 mostra um índice para os dados de teste da Fig. 2.2 combinando os campos CITY e STATUS (nessa ordem).

CITY/STATUS	Indicadores de localização
Athens/30	↑ S5
London/20	↑ S1, ↑ S4
Paris/10	↑ S2
Paris/30	↑ S3

Fig 2.13 Indexando em CITY/STATUS.

Com estes índices nós podemos responder a consultas do tipo “Encontre todos os fornecedores em Paris com status 10”. Se o índice combinado não existisse, esta consulta envolveria (a) encontrar todos os fornecedores em Paris, (b) encontrar todos os fornecedores com status 10, e (c) extrair os fornecedores comuns às duas listas. Além disso, como as etapas (a) e (b) poderiam ser perfeitamente executadas na ordem inversa, nós teríamos um problema de estratégia (isto é, o que fazer primeiro).

Note que o índice combinado também é um índice do campo CITY, visto que todas as entradas de uma dada cidade são pelo menos consecutivas dentro do índice. (Teria que ser preparado um índice separado se fosse requerida indexação em STATUS, no entanto.) Em geral, um índice que seja a combinação de n campos F_1, F_2, \dots, F_n (nessa ordem) servirá também como um índice de F_1 , como um índice da combinação $F_1 F_2$ (ou $F_2 F_1$), como um índice da combinação $F_1 F_2 F_3$ (em qualquer ordem), e assim por diante. Portanto, o número total de índices requeridos para prover uma indexação completa desta maneira não é tão grande como poderia parecer à primeira vista.

Seletividade no índice

É preciso que o leitor compreenda que não é necessário prover um acesso via índice para cada ocorrência de registro no arquivo indexado. Em algumas situações pode ser útil ter entradas no índice somente para valores selecionados do campo indexado. Por exemplo, consideremos um arquivo de empregados no qual 95 por cento desses empregados tenham o status 1, e 5 por cento tenham o status 2. Dificilmente seria produtivo existir um índice

geral para o campo status. Por outro lado, poderia ser útil haver um índice indicando todos os empregados de status 1 — em outras palavras, selecionar um subconjunto do conjunto de possíveis valores a serem indexados.

Técnicas de compressão

Técnicas de compressão são maneiras de se reduzir a quantidade de memória requerida para um determinado conjunto de dados armazenados. Uma técnica comum é a de se substituir cada valor individual de dado por alguma representação da diferença entre ele e o valor do dado próximo a ele. Esta técnica pode ser usada sempre que o acesso aos dados se fizer *de forma seqüencial* — por exemplo, dentro de um índice ou dentro de um “grupo” de um arquivo no qual exista indexação não densa (veja seção 2.3).

Para consolidar o entendimento, vamos tomar como exemplo um único bloco de um índice de um arquivo de empregados. (Na prática, as técnicas de compressão são muito mais freqüentemente usadas em índices do que em arquivos de dados.) Vamos supor que as quatro primeiras entradas neste bloco sejam dos seguintes empregados

ROBERTON
ROBERTSON
ROBERTSTONE
ROBINSON

Os nomes dos empregados têm 12 caracteres de comprimento (de tal forma que cada um desses nomes pode ser considerado — na sua forma sem compressão — como possuindo à direita um número apropriado de brancos).

A primeira técnica de compressão que podemos aplicar é a de substituir os caracteres iniciais de cada entrada que são iguais aos da entrada anterior por uma contagem correspondente: *compressão frontal*. Isto nos dá

0-ROBERTONbbbb
6-S0bbb
7-TONEb
3-INSONbbbb

(os brancos finais estão agora explicitamente mostrados como b).

A segunda técnica de compressão é a de *compressão posterior*. Uma forma de se aplicar compressão posterior no exemplo seria eliminar-se todos os brancos finais (substituindo-os por um contador apropriado). Pode ser obtida uma compressão adicional retirando-se todos os caracteres à direita do necessário para distinguir uma entrada das duas que lhe são adjacentes. Isto nos daria

0-7-ROBERTO
6-2-S0
7-1-T
3-1-I

onde o segundo número em cada entrada é o contador do número de caracteres gravado. (Nós assumimos que a próxima entrada não possui ROBI como seus primeiros quatro ca-

racteres em sua forma sem compressão.) Note, entretanto, que na realidade nós perdemos alguma informação deste índice. Isto é, após a descompressão o aspecto será este:

ROBERTO?????
ROBERTSO????
ROBERTST????
ROBI?????????

(onde? significa um caracter desconhecido). Isto só é permissível quando os valores completos encontram-se gravados *em algum outro lugar* (neste caso, no arquivo indexado).

Representação por indicador de localização simbólico

Até agora nós partimos do princípio de que todos os indicadores de localização são SRAs. No entanto, como mencionado na Seção 2.1, os valores de SRA podem se alterar quando um arquivo armazenado é reorganizado. Quando isto ocorre na reorganização de um arquivo armazenado, todos os arquivos que possuem indicadores voltados para ele — em particular, todos os índices — terão que ser atualizados para passarem a conter os novos valores de SRA. Por isso, a reorganização pode se tornar um processo custoso e demorado. Entretanto, isto pode ser evitado pelo uso de *indicadores de localização simbólicos*. Isso é, os indicadores de localização com valores de SRA podem ser substituídos pelos valores correspondentes das chaves primárias. Por exemplo, uma entrada no índice para o fornecedor S1 conteria agora não o endereço (SRA) do registro S1, mas o valor real do dado 'S1'. Este valor pode então ser usado para se localizar o registro correspondente (supondo que sempre seja possível ter-se acesso direto a um registro na base de sua chave primária). Claramente um índice que use chaves simbólicas não precisa ser atualizado em função de uma reorganização do arquivo indexado (o valor simbólico 'S1' ainda identifica o registro do fornecedor S1, mesmo que este registro tenha mudado de lugar e seu SRA tenha se modificado). Naturalmente, o acesso via este índice será mais lento do que um acesso via um índice que use indicadores de localização diretos.

As Figs. 2.14 e 2.15 mostram o resultado da aplicação desta técnica às representações das Figs. 2.4 (indexação em CITY) e 2.7 (organização invertida), respectivamente. Na Fig. 2.15 note que desapareceu totalmente a necessidade do arquivo armazenado de fornecedores.

Arquivo CITY

CITY	S=
Athens	S5
London	S1
	S4
Paris	S2
	S3

Arquivo FORNECEDORES

S=	SNAME	STATUS
S1	Smith	20
S2	Jones	10
S3	Blake	30
S4	Clark	20
S5	Adams	30

Fig. 2.14 Indexação em CITY (indicadores de localização simbólicos).

Índice SNAME	
SNAME	S#=
Smith	S1
Jones	S2
Blake	S3
Clark	S4
Adams	S5

Índice STATUS	
STATUS	S#=
10	S2
20	S1, S4
30	S3, S5

Índice CITY	
CITY	S#=
Athens	S5
London	S1, S4
Paris	S2, S3

Fig. 2.15 Organização invertida (indicadores de localização simbólicos).

EXERCÍCIOS

Os exercícios 2.1 – 2.3 podem ser adequados como base para uma discussão em grupo; são voltados para considerações mais profundas sobre vários problemas de projeto. Já os exercícios 2.5 e 2.6 são mais de base matemática.

2.1 O banco de dados de uma companhia contém informações sobre as divisões, departamentos e empregados dessa companhia. Cada empregado trabalha em um departamento; cada departamento é parte de uma divisão. Crie alguns dados de exemplo e projete algumas estruturas de armazenamento possíveis para estes dados. Quando possível, mostre as vantagens relativas de cada estrutura – isto é, considere como as operações típicas de recuperação e atualização seriam feitas em cada caso. *Sugestão:* As exigências “cada empregado trabalha em um departamento” e “cada departamento está em uma divisão” são estruturalmente semelhantes à exigência “cada fornecedor tem uma cidade”. (São todos exemplos de relacionamentos vários-para-um.) Uma diferença existente entre este exercício e o exemplo fornecedor-cidade é que nós provavelmente gostaríamos de registrar mais informação no banco de dados para departamentos e divisões do que fizemos para cidades.

2.2 Repita o exercício 2.1 considerando um banco de dados que contenha informações sobre clientes e itens. Cada cliente pode pedir qualquer quantidade de itens diferentes; cada item pode ser pedido por qualquer quantidade de clientes. *Sugestão:* há um relacionamento de vários-para-muitos entre clientes e itens. Uma forma de se representar tal relacionamento é por meio do *duplo índice*. Um duplo índice é um índice usado para indexar simultaneamente dois arquivos de dados. Uma determinada entrada corresponde a um *par* de registros de dados relacionados, um de cada um dos dois arquivos, e contém dois valores indexados e dois indicadores de localização. Você pode imaginar outras maneiras de representar relacionamentos vários-para-muitos (-para-muitos...)?

2.3 Repita o exercício 2.1 para um banco de dados que contenha informações sobre peças e componentes (um componente em si mesmo é uma peça e pode ter componentes adicionais). *Sugestão:* Como este problema difere daquele do exercício 2.2?

2.4 Os primeiros dez valores de um campo indexado de um determinado arquivo indexado são como se segue

```

ABRAHAMS, GK
ACKERMANN, LZ
ACKROYD, S
ADAMS, T
ADAMS, TR
ADAMSON, CR
ALLEN, S
AYRES, ST
BAILEY, TE
BAILEYMAN, D

```

(Cada um deles está completado com brancos à direita até um comprimento total de 15 caracteres.) Mostre os valores realmente gravados no *índice* se as técnicas de compressão descritas na Seção 2.4 forem aplicadas. Qual o percentual de economia de espaço? Mostre as etapas envolvidas na recuperação (ou tentativa de recuperação) das ocorrências de registros armazenados ACKROYD,S e ADAMS,V. Mostre também as etapas envolvidas na inserção da ocorrência de registro armazenado ALLINGHAM,M.

2.5 Suponhamos que temos um índice em múltiplos níveis tal que o nível mais baixo contém uma entrada para cada uma das N ocorrências de registros armazenados, e que cada nível acima desse contém uma entrada para cada bloco do nível inferior. Suponhamos também que cada bloco do índice contém n entradas, e que o índice se estende até um bloco único no nível mais alto. Derive expressões para o número de *níveis* do índice e para o número de *blocos* do índice.

2.6 Vamos definir "indexação completa" como significando que existe um índice para cada combinação distinta de campos no arquivo indexado. Quantos índices seriam necessários para prover uma indexação completa de um arquivo definido com (a) 3 campos; (b) 4 campos; (c) N campos?

REFERÊNCIAS E BIBLIOGRAFIA

Estas referências estão organizadas em grupos, como se segue: [2.1–2.5] são livros-texto, ou inteiramente devotados ao tópico deste capítulo ou incluindo pelo menos um tratamento detalhado dele. [2.6–2.8] são estudos. [2.9] e [2.10] são artigos bastante antigos mas ainda assim relevantes. [2.11] e [2.12] discutem endereçamento randômico, [2.13–2.15] índices combinados, [2.16–2.18] outras técnicas de indexação (incluindo árvores-B), e [2.19–2.21] técnicas de compressão. [2.22–2.24] são ensaios de desenvolvimentos de bases teóricas sobre o assunto. Finalmente, [2.25–2.28] discutem algumas técnicas diversas.

Veja também [1.16].

2.1 D. E. Knuth. *The Art of Computer Programming. Vol. III: Sorting and Searching*. Reading, Mass.: Addison-Wesley (1973).

Inclui uma análise abrangente de algoritmos de pesquisa (Capítulo 6). Para pesquisa em *bancos de dados*, onde o dado reside em memória secundária, as seções mais diretamente aplicáveis são 6.2.4 (Multiway Tress), 6.4 (Hashing), e 6.5 (Retrieval on Secondary Keys).

2.2 D. Lefkovitz. *File Structures for On-Line Systems*. Rochelle Park, N.J.: Spartan Books (1969).

2.3 J. Martin. *Computer Data-Base Organization*. Englewood Cliffs, N.J.: Prentice-Hall (1975).

Este livro está dividido em duas grandes partes: Organização Lógica e Organização Física. A parte II consiste de uma extensa descrição (mais de 300 páginas) sobre estruturas de armazenamento e estratégias de acesso associadas.

2.4 G. Wiederhold. *Database Design*. New York: McGraw-Hill (1977).

Este livro (15 capítulos) inclui um bom levantamento sobre dispositivos de memória secundária e seus parâmetros de desempenho (um capítulo, cerca de 40 páginas) e uma análise abrangente das correspondentes estruturas de armazenamento (três capítulos, mais de 200 páginas).

2.5 S. P. Ghosh. *Data Base Organization for Data Management*. New York: Academic Press (1977).

Um tratamento bastante formal.

2.6 G. G. Dodd. "Elements of Data Management Systems". *ACM Comp. Surv.* 1, nº 2 (junho de 1969).

2.7 W. D. Maurer and T. G. Lewis. "Hash Table Methods". *ACM Comp. Surv.* 7, nº 1 (março de 1975).

2.8 D. Comer. "The Ubiquitous B-tree". *ACM Comp. Surv.* 11, nº 2 (junho de 1979).

2.9 W. W. Peterson. "Addressing for Random-Access Storage". *IBM J. R. & D* 1, nº 2 (abril de 1957).

2.10 W. Buchholz. "File Organization and Addressing". *IBM Sys. J.* 2, nº 2 (junho de 1963).

2.11 R. Morris. "Scatter Storage Techniques". *CACM* 11, nº 1 (janeiro de 1968).

Este artigo concentra-se basicamente na forma como se aplicam as técnicas de randomização à tabela de símbolos de um Assembler ou compilador. Entretanto, é um levantamento útil das

técnicas disponíveis, podendo ser valioso para quem esteja interessado nos problemas de projetos de estruturas de armazenamento.

- 2.12 V. Y. Lum, P. S. T. Tuen, and M. Dodd. "Key-to-Address Transform Techniques: A Fundamental Performance Study on Existing Formatted Files". *CACM* 14, nº 4 (abril de 1971).

Uma investigação sobre o desempenho de diversos algoritmos de randomização diferentes. A conclusão é que o método divisão/resto parece ser o que apresenta melhor desempenho global.

- 2.13 V. Y. Lum. "Multi-attribute Retrieval with Combined Indexes". *CACM* 13, nº 11 (novembro de 1970).

Este artigo introduziu a técnica de indexação por combinação de campos.

- 2.14 J. K. Mullin. "Retrieval-Update Speed Tradeoffs Using Combined Indices". *CACM* 14, nº 12 (dezembro de 1971).

Uma continuação do [2.13] que fornece estatísticas de desempenho para o esquema de índice combinado para várias razões de recuperação/memória.

- 2.15 B. Shneiderman. "Reduced Combined Index for Efficient Multiple Attribute Retrieval". *Information Systems* 2, nº 4 (1976).

Propõe um refinamento à técnica de indexação combinada de Lum [2.13] reduzindo consideravelmente as sobrecargas de espaço de armazenamento e tempo de pesquisa. Por exemplo, o índice combinado ABCD, BCDA, CDAB, DABC, ACBD, BDAC – veja a resposta do exercício 2.6 (b) – poderia ser substituído pela combinação ABCD, BCD, CDA, DAB, AC, BD. Se cada um dos A, B, C, D pode assumir 10 valores distintos, então no pior caso a combinação original envolveria 60.000 entradas de índice, e a combinação reduzida 13.200 entradas.

- 2.16 R. Bayer and C. McCreight. "Organization and Maintenance of Large Ordered Indexes". *Acta Informatica* 1, nº 3 (1972).

- 2.17 R. E. Wagner. "Indexing Design Considerations". *IBM Sys. J.* 12, nº 4 (1973).

Uma boa descrição dos conceitos de indexação, com detalhes das técnicas – inclusive algoritmos de compressão – usados no Virtual Storage Access Method, VSAM, da IBM.

- 2.18 M. R. Vose and J. S. Richardson. "An Approach to Inverted Index Maintenance". *BCS Comp. Bull.* 16, nº 5 (maio de 1972).

Descreve uma abordagem de múltiplas listas para a construção e manutenção de índices secundários. Este método evita alguns dos problemas de entradas de índices com comprimento variável. Cada ocorrência de registro armazenado é representada dentro do sistema por um "número de índice seqüencial" (SIN), designado por ocasião do primeiro armazenamento da ocorrência. Os valores SIN são designados em seqüência ascendente. O "índice básico" contém uma entrada para cada ocorrência de registro de dados, fornecendo o seu SIN e o seu endereço físico, e é neste índice que é usada a organização em listas múltiplas. Para cada campo indexado, cada entrada no índice básico fornece o SIN da próxima ocorrência que possui o mesmo valor naquele campo. Para cada valor de cada campo indexado, são mantidos indicadores das localizações do início e do fim da cadeia dentro do índice básico.

- 2.19 B. A. Marron and P. A. D. de Maine. "Automatic Data Compression". *CACM* 10, Nº 11 (novembro de 1967).

Fornecer dois algoritmos de compressão/descompressão: NUPAK, que opera com dados numéricos, e ANPAK, que opera com dados alfanuméricos ou "qualsquer" (isto é, qualquer seqüência de bits).

- 2.20 D. A. Huffman. "A Method for the Construction of Minimum Redundancy Codes". *Proc. IRE* 40 (setembro de 1952).

A codificação de Huffman é uma técnica de codificação em caractere que pode resultar em uma compressão significativa de dados, se não houver a ocorrência de todos os caracteres com igual freqüência. A idéia básica é a de designar codificações de seqüências de bits para representarem os caracteres, de tal forma que caracteres diferentes são representados por seqüências de comprimentos diferentes, sendo os caracteres de ocorrência mais comum representados pelas seqüências mais curtas. Também, nenhum caractere possui uma codificação (digamos de n bits)

tal que aqueles *n bits* sejam idênticos aos *n* primeiros *bits* de alguma outra codificação de caractere. Como um exemplo simples, suponhamos que o dado a ser representado envolva somente os caracteres A, B, C, D, E, e suponhamos que a freqüência relativa de ocorrência desses caracteres seja como a dada na seguinte tabela (segunda coluna).

Caráter	Freqüência	Código
E	35%	1
A	30%	01
D	20%	001
C	10%	0001
B	5%	0000

O carácter E apresenta a freqüência mais alta e por isso a ele é designado o código mais curto, digamos 1 *bit*. Então todos os outros códigos terão que começar com o *bit* 0, e ter pelo menos dois *bits* de comprimento (um *bit* zero sozinho não seria válido, pois não poderia ser distinguido do *bit* inicial dos outros códigos). Ao carácter A é designado o próximo código mais curto, digamos 01; aos caracteres D, C, e B são designados respectivamente 001, 0001, 0000 (veja a tabela). Com estas designações, o comprimento médio esperado de um carácter codificado, em bits, é

$$0.35 * 1 + 0.30 * 2 + 0.20 * 3 + 0.10 * 4 + 0.05 * 4 = 2.15 \text{ bits},$$

enquanto que se a cada carácter tivesse sido designado o mesmo número de *bits*, como no esquema convencional de codificação de caracteres, nós precisaríamos de três *bits* por carácter (para permitir cinco caracteres diferentes).

2.21 H. K. Chang. "Compressed Indexing Method". *IBM Technical Disclosure Bulletin II*, nº 11 (abril de 1969).

2.22 D. Hsiao and F. Harary. "A Formal System for Information Retrieval from Files". *CACM* 13, nº 2 (fevereiro de 1970).

Este artigo procura unificar as idéias das várias estruturas de armazenamento – organização invertida, organização seqüencial indexada, organização em múltiplas listas, etc. – em uma “estrutura geral de arquivo”, e assim formar uma base para a teoria das estruturas de armazenamento. É apresentado um “algoritmo geral de recuperação” para recuperar as ocorrências, satisfazendo uma combinação Booleana arbitrária de “atributo = valor” de condições, a partir da estrutura geral.

2.23 D. G. Severance. "Identifier Search Mechanisms: A Survey and Generalized Model." *ACM Comp. Surv.* 6, nº 3 (setembro de 1974)

2.24 M. E. Senko, E. B. Altman, M. M. Astrahan, and P. L. Fehder. "Data Structures and Accessing in Data-Base Systems". *IBM Sys J.* 12, nº 1 (1973).

Este artigo possui três partes:

I. Evolução dos Sistemas de Informação

II. Organização da Informação

III. Representações de Dados e o Modelo de Acesso Independente de Dados.

A parte I consiste de um curto levantamento histórico sobre o desenvolvimento dos sistemas de bancos de dados. A parte II descreve o modelo do conjunto entidade, que corresponde ao nível conceitual da arquitetura ANSI/SPARC. A parte III forma uma introdução ao Modelo de Acesso Independente de Dados (DIAM), que é uma tentativa de descrever o banco de dados em termos de quatro níveis sucessivos de abstração: níveis: conjunto entidade (o mais alto), série, codificação e dispositivo físico. Esses quatro níveis podem ser vistos como uma definição mais detalhada, mas ainda abstrata, das porções mais baixas da arquitetura da Fig. 1.5. Dos quatro níveis, nós podemos caracterizar os três mais baixos como se segue.

- *Nível de série*. Os caminhos de acesso ao dado estão definidos como conjuntos ordenados ou “séries” de objetos dados. Estão identificados três tipos de séries: séries atômicas (exemplo:

uma série ligando valores de campos para formar uma ocorrência de registro armazenado de PEÇAS), séries entidades (exemplo: uma série ligando as ocorrências de registros PEÇAS de peças vermelhas) e séries de ligação (exemplo: uma série ligando uma ocorrência de registro FORNECEDOR com ocorrências de PEÇAS para as peças fornecidas por aquele fornecedor).

- Nível de codificação. Objetos dados e séries são mapeados em espaços lineares de endereçamento, usando uma representação única de primitiva simples conhecida como unidade básica de codificação.
- Nível de dispositivo físico. São designados espaços lineares de endereçamento para formatar subdivisões físicas do meio real de gravação.

O objetivo do DIAM é fornecer uma estrutura de base na qual possam ser verificados diversos problemas de projeto de uma forma controlada e estruturada.

2.25 D. G. Severance and G. M. Lohman. "Differential Files: Their Application to the Maintenance of Large Databases". *ACM Transactions on Database Systems* 1, nº 3 (setembro de 1976).

Este artigo descreve "arquivos diferenciais" e discute suas vantagens. A idéia básica é a de que as atualizações não sejam feitas diretamente no próprio banco de dados – ao invés disso, sejam gravadas em um arquivo fisicamente distinto (o arquivo diferencial), e em um momento subsequente adequado sejam introduzidas no banco de dados. São apontadas as seguintes vantagens para esta abordagem. Note que as seis primeiras destas se referem à integridade e recuperação do banco de dados, enquanto que as quatro restantes são vantagens operacionais.

- Os custos para descarga do banco de dados são reduzidos
- Fica facilitada a descarga incremental
- A descarga e a reorganização podem ambas ser executadas juntamente com operações de atualização
- A recuperação após erro de programa é mais rápida
- A recuperação após falha de *hardware* é mais rápida
- Fica reduzido o risco de uma perda séria de dados
- "Arquivos memo" são suportados eficientemente
- Fica simplificado o desenvolvimento de *software*
- Fica simplificado o *software* do arquivo principal
- Podem ser reduzidos custos futuros de memória

Um problema não discutido é o de como suportar acesso seqüencial por chave aos dados quando alguns registros estão no banco de dados real e alguns no arquivo diferencial.

2.26 B. J. Dzubak and C. R. Warburton. "The Organization of Structured Files". *CACM* 8, nº 7 (julho de 1965).

Compara e contrasta dez métodos de armazenamento e acesso a um "arquivo estruturado" (isto é, uma coleção de informações na forma de um gráfico; por exemplo, a explosão de uma peça).

2.27 E. Wong and T. C. Chiang. "Canonical Structure in Attribute Based File Organization". *CACM* 14, nº 9 (setembro de 1971).

É proposta uma nova estrutura que possui várias vantagens. Parte do princípio de que todas as solicitações de recuperação sejam expressas como uma combinação Booleana das condições elementares "campo = valor", e de que essas condições elementares são conhecidas. Então o arquivo pode ser separado em subconjuntos para finalidades de armazenamento; os subconjuntos são os átomos da álgebra Booleana que é o conjunto de todos os conjuntos de ocorrências de registros recuperáveis via solicitações de acesso Booleanas. As vantagens desta técnica incluem o seguinte:

- a) Nunca é necessária a interseção de conjuntos (de átomos)
- b) Uma solicitação arbitrária pode facilmente ser convertida em uma solicitação de (a união de) um ou mais átomos.
- c) Esta união nunca envolve eliminação de duplicatas.

Uma desvantagem dos esquemas de randomização em geral é que à medida que o banco de dados cresce, também o número de colisões cresce, e o tempo médio de acesso aumenta correspondente (pois mais e mais tempo é gasto na execução das pesquisas seqüenciais através dos conjuntos de colisões). Este artigo descreve uma variante elegante da técnica básica de randomização, na qual fica garantido que nunca serão necessários mais do que dois acessos à memória secundária para localizar o registro correspondente a uma determinada chave. Segue-se uma descrição resumida deste esquema.

1. Seja h a função básica de randomização, e seja K o valor da chave primária de algum registro R . Randomizando K (isto é, avaliando $h(k)$, nós obtemos um valor K' , que chamamos *pseudochave* de R . As pseudochaves não são interpretadas diretamente como endereços, mas sim como indicadores de registros armazenados de uma forma algo indireta, como descrito abaixo.
2. O arquivo armazenado possui um *diretório* associado a ele (também mantido na memória secundária). O diretório consiste de um cabeçalho, contendo um valor d chamado de profundidade do diretório, e uma lista de 2^d indicadores de localização. Estes indicadores de localização indicam os blocos de dados, que contêm os registros armazenados (vários registros por bloco). Se considerarmos os primeiros d bits de uma pseudochave com um inteiro binário b não designado, então o i -ésimo indicador de localização no diretório ($1 \leq i \leq 2^d$) indica o bloco de dados que contém todos os registros para os quais b assume o valor $i - 1$. Em outras palavras, o primeiro indicador de localização indica o bloco que contém todos os registros para os quais b é todo zeros; o segundo indica o bloco contendo todos os registros para os quais b é $0 \dots 01$, e assim sucessivamente. (Estes 2^d indicadores de localização não são tipicamente todos distintos; isto é, tipicamente haverá menos do que 2^d blocos de dados distintos.) Portanto, para encontrar o registro possuindo a chave K , nós randomizamos K para encontrar a pseudochave K' e tomamos os primeiros d bits dessa pseudochave; se esses bits tiverem o valor numérico $i - 1$, nós vamos para a i -ésima entrada no diretório (um acesso na memória secundária) e seguimos o indicador de localização que encontrarmos ali, que nos levará ao bloco de dados contendo o registro desejado (mais um acesso à memória secundária). Note que a organização interna dos registros dentro de um bloco de dados é independente do esquema de randomização extensível. Ele pode perfeitamente envolver uma operação interna de randomização. Mas encontrar um registro *dentro* de um bloco não requer nenhum acesso adicional à memória secundária, uma vez que o bloco já foi lido para a memória principal.
3. Cada bloco de dados também possui um cabeçalho dando a *profundidade local p* daquele bloco ($p \leq d$). Suponhamos, por exemplo, que $d = 3$, e que o primeiro indicador de localização no diretório (o indicador 000) indique um bloco de dados no qual a profundidade local p é de 2. Profundidade local 2 significa neste caso que este bloco contém não somente todos os registros cujas pseudochaves começam por 000, mas que contém *todos* os registros com pseudochaves começando por 00 (isto é, as que começam por 000 e as que começam por 001). Em outras palavras, o indicador de localização 001 do diretório também indica este bloco de dados.
4. Continuando o exemplo de (3) acima, suponhamos que o bloco de dados está cheio, e que nós desejamos agora inserir um novo registro tendo uma pseudochave começando por 000 (ou 001). Neste momento o bloco de dados é dividido em dois; isto é, é obtido um bloco novo e vazio, e todos os registros 001 são movidos do bloco antigo para o novo. O indicador de localização 001 no diretório é atualizado para indicar o novo bloco (o indicador de localização 000 continua indicando o bloco antigo).
5. Ainda continuando com o exemplo, suponhamos que o bloco de dados 000 se torne cheio novamente e tenha que ser de novo dividido. O diretório existente não pode manusear a divisão, porque a profundidade do bloco a ser dividido já é igual à profundidade do diretório. Portanto nós "dobramos o diretório"; isto é, aumentamos d de 1 e substituímos cada indicador de localização por um par de indicadores de localização adjacentes idênticos. O bloco de dados pode agora ser dividido; os registros 0000 são deixados no bloco antigo e os registros 0001 vão para o novo bloco; o primeiro indicador de localização no diretório não é modificado (isto é ainda indica o bloco antigo) e o segundo indicador de localização é mudado para indicar o novo bloco. Podemos notar que dobrar o diretório é uma operação muito barata, pois não envolve qualquer acesso aos blocos de dados.

3

Estruturas de Dados e Operadores Correspondentes

3.1 INTRODUÇÃO

Neste capítulo nós vamos examinar o problema de como o banco de dados deve parecer ao usuário. Como explicamos no capítulo 1, é normal apresentar-se ao usuário uma visão dos dados na qual detalhes de como o dado está representado no armazenamento são deliberadamente omitidos. Esta visão é a *visão externa*. (Na maioria dos sistemas atuais as visões externa e conceitual são bastante semelhantes, se não idênticas; para os sistemas que não fazem distinção entre as duas, os itens a seguir aplicam-se a ambos os níveis.)

A gama de estruturas de dados suportada ao nível de usuário (externo ou conceitual) é um fator que afeta de maneira crítica muitos componentes do sistema. Em particular, dita o projeto das linguagens de manipulação de dados correspondentes, pois cada DML têm que ter sua operação definida em termos dos seus efeitos sobre aquelas estruturas de dados. Portanto a questão “Que estruturas de dados e operadores associados deve o sistema suportar?” é crucial. Nós podemos categorizar de maneira conveniente os sistemas de bancos de dados de acordo com a abordagem por eles adotada na resposta a esta questão central. As três abordagens mais conhecidas são:

- A abordagem relacional
- A abordagem hierárquica
- A abordagem em rede

Este fato pesou na estrutura geral deste livro – as partes 2, 3 e 4 consistem de exames detalhados de cada uma destas três abordagens. O objetivo do capítulo atual é o de abrir caminho para esses capítulos futuros, fornecendo uma breve introdução e uma comparação entre as três abordagens. Por isso este capítulo pode ser encarado como sendo chave para todo o livro; entretanto, o leitor deve estar prevenido de que não tentaremos ser rigorosos ou completos até que tenhamos chegado a capítulos mais avançados.

3.2 A ABORDAGEM RELACIONAL

Esta seção e as duas seguintes estão baseadas em um banco de dados de teste contendo fornecedores, peças e embarques. A Fig. 3.1 mostra os dados de teste em forma relacional; isto é, ela representa uma *visão relacional* dos dados.

Pode-se ver que os dados estão organizados em três tabelas: S (fornecedores), P (peças) e SP (embarques). A tabela S contém, para cada fornecedor, um número de fornecedor, nome, código de status, e localização; a tabela P contém, para cada peça, um número de peça, nome, cor, peso e a localização de onde a peça está armazenada; e a tabela SP contém, para cada embarque, um número de fornecedor, um número de peça e a quantidade embarcada. Incidentalmente, a tabela S consiste das primeiras três linhas da tabela da Fig. 2.2, que nós podemos observar ser uma visão relacional dos dados usados como base para os exemplos do capítulo 2. Vamos fazer as mesmas suposições sobre os fornecedores que fizemos no capítulo 2, ou seja, que cada fornecedor possui um único número e exatamente um nome, valor de status, e localização. Semelhantemente, vamos supor que cada peça possui um único número de peça e exatamente um nome, cor, peso e localização; e que, em determinado momento, não há mais do que um embarque para uma determinada combinação fornecedor/peça.

Cada uma das três tabelas se assemelha bastante a um arquivo seqüencial convencional, com as linhas da tabela correspondendo aos registros do arquivo e as colunas correspondendo aos campos dos registros.

A tabela P, por exemplo, contém quatro linhas ou registros, cada uma consistindo de cinco campos. Entretanto, há certas diferenças significativas – a serem explicadas em detalhes no capítulo 4 – entre tabelas como as da Fig. 3.1 e os arquivos seqüenciais tradicionais. Cada uma dessas tabelas é na realidade um caso especial das construções conhecidas em matemática como uma *relação* – um termo que tem definição muito mais precisa do que o termo mais tradicional de processamento de dados “*arquivo*”, ou “*tabela*” nesse caso. (Entretanto, neste livro nós usaremos freqüentemente “*tabela*” como um sinônimo de “*relação*”.) A abordagem relacional aos dados está baseada na observação de que arquivos que obedecem a certas limitações podem ser considerados como relações matemáticas, e consequentemente a teoria elementar de relações pode ser usada para lidar com vários problemas práticos com os dados desses arquivos.

S	S#	SNAME	STATUS	CITY
S1	Smith	20	London	
S2	Jones	10	Paris	
S3	Blake	30	Paris	

P	P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London	
P2	Bolt	Green	17	Paris	
P3	Screw	Blue	17	Rome	
P4	Screw	Red	14	London	

SP	S#	P#	QTY
S1	P1	300	
S1	P2	200	
S1	P3	400	
S2	P1	300	
S2	P2	400	
S3	P2	200	

Fig. 3.1 Dados de teste em forma relacional

Por isso, na maior parte da literatura relacional, tabelas como as da Fig. 3.1 são tratadas como relações. As linhas dessas tabelas são usualmente conhecidas como tuplas (usualmente pronunciado, em inglês, para rimar com “couples” [pares]), de novo pelo fato de ter este termo uma definição mais precisa do que “linha” ou “registro”. Neste livro, entretanto, nós estaremos usando alternativamente os termos “tupla” e “linha”. De maneira semelhante, as colunas são usualmente conhecidas como atributos; também neste caso usaremos os dois termos alternativamente.

Um conceito que a teoria relacional enfatiza e para o qual não parece haver um termo estabelecido em processamento de dados é o conceito de domínio. Um domínio é um reservatório de valores do qual são retirados os que aparecem em uma determinada coluna. Por exemplo, os valores que aparecem na coluna P# tanto na tabela P quanto na tabela SP foram retirados do domínio básico de todos os números válidos de peças. Este domínio em si, embora não possa ser explicitamente gravado no banco de dados como um conjunto real de valores, estará definido no esquema apropriado e terá seu próprio nome; colunas baseadas neste domínio poderão ter ou não o mesmo nome. (Obviamente elas deverão ter nomes diferentes caso pudesse vir a ocorrer uma ambigüidade; por exemplo – se duas colunas da mesma tabela fossem retiradas de um mesmo domínio. Um exemplo dessa situação será dado no capítulo 4.)

Observe que as relações S e SP têm um domínio (números de fornecedores) em comum; da mesma forma P e SP (nímeros de peças), e também S e P (localizações). Um aspecto crucial da estrutura de dados relacional é que *associações entre tuplas (linhas) são representadas somente por valores de dados em colunas retiradas de um domínio comum*. O fato de estarem o fornecedor S3 e a peça P2 localizados na mesma cidade, por exemplo, é representado pelo aparecimento do mesmo valor na coluna CITY para as duas tuplas envolvidas. De fato, é uma característica da abordagem relacional que todas as informações no banco de dados – tanto “entidades” como “relacionamentos”, para usar a terminologia da Seção 1.2 – estejam representadas de uma maneira uniforme, ou seja, na forma de tabelas. Como nós veremos mais tarde, esta característica não é compartilhada pelas abordagens hierárquica e em rede.

Nós não vamos discutir aspectos adicionais da estrutura relacional neste momento; encontraremos um tratamento mais completo no Capítulo 4. Fica claro, entretanto, que a estrutura relacional é muito fácil de ser entendida. Mas a simplicidade de representação dos dados não é tudo. Do ponto de vista do usuário, a linguagem de manipulação dos dados – isto é, o conjunto de operadores fornecidos para manipular os dados representados na forma relacional – é no mínimo tão importante quanto. Antes de discutirmos operadores relacionais em detalhe, observemos que a uniformidade da representação dos dados leva a uma correspondente uniformidade no conjunto de operadores: Uma vez que a informação é representada de uma e somente uma maneira, nós precisamos de apenas um operador para cada uma das funções básicas (inserção, remoção etc.) que desejarmos executar. Isto contrasta com a situação de estruturas mais complexas, onde a informação pode ser representada de diversas maneiras, e consequentemente são exigidos diversos conjuntos de operadores. Como nós veremos na Parte 4 deste livro, por exemplo, o DBTG do sistema baseado em rede fornece dois operadores para “inserção”: STORE, para criar uma ocorrência de registro, e CONNECT para criar um “elo” entre duas ocorrências de registros.

Vamos considerar agora algumas operações específicas. Para recuperação, o operador básico de que precisamos é “get next where” (obtenha o próximo no qual), que irá trazer a próxima linha de uma tabela satisfazendo a alguma condição especificada. “Pró-

ximo” é interpretado em relação à *posição corrente* (normalmente a linha cujo acesso foi o mais recente: para o caso inicial vamos supor estar imediatamente antes da primeira linha da tabela). Este operador está ilustrado nos exemplos fornecidos pela Fig. 3.2, que mostra um esboço da codificação requerida para o manuseio de duas consultas específicas ao banco de dados da Fig. 3.1. As duas consultas são intencionalmente simétricas (cada uma é o inverso da outra).

Como para as outras operações, vamos nos satisfazer em considerar três problemas simples, um para cada uma das funções básicas inserção, remoção e atualização,¹ indicando brevemente em cada caso uma operação relacional possível para o manuseio.

Inserção Dada uma informação referente a um novo fornecedor S4 na área W, insira-a no banco de dados.

- Insira a tupla de W na relação S.

Remoção Remova o embarque ligando a peça P2 com o fornecedor S3.

- Remova a tupla SP na qual P# = P2 e S# = S3.

Atualização O fornecedor S1 mudou-se de Londres para Amsterdam.

- Atualize a tupla S onde S# = S1 ajustando CITY para Amsterdam.

Nós retornaremos à questão dos operadores relacionais na Seção 3.5. Antes disso, entretanto, vamos tecer considerações sobre as abordagens hierárquica e em rede.

Q1: Encontre os números dos fornecedores que fornecem a peça P2.	Q2: Encontre os números das peças fornecidas pelo fornecedor S2.
<pre> do until no more shipments; get next shipment where P# = P2; print S#; end; </pre>	<pre> do until no more shipments; get next shipment where S# = S2; print P#; end; </pre>

Fig. 3.2 Dois exemplos de consultas à visão relacional.

3.3 A ABORDAGEM HIERÁRQUICA

A Fig. 3.3 mostra uma possível *visão hierárquica* do banco de dados de fornecedores e peças. Nesta visão, o dado está representado por uma estrutura em árvore simples, com as peças acima dos fornecedores. O usuário vê quatro árvores separadas, ou ocorrências hierárquicas, uma para cada peça. Cada árvore consiste de uma ocorrência de registro de peça, juntamente com um conjunto de ocorrências de registros de fornecedores subordinados, um para cada fornecedor da peça. Cada ocorrência de fornecedor inclui a quantidade

¹ Aqui “atualização” está sendo usada como um sinônimo de “modificação”, ao invés de como um termo genérico que inclua “inserção” e também “remoção”. A literatura sobre bancos de dados é muito inconsistente no uso do termo “atualização” e este livro não é uma exceção. Consequentemente nós iremos confiar no contexto para tornar mais claro o significado desejado.

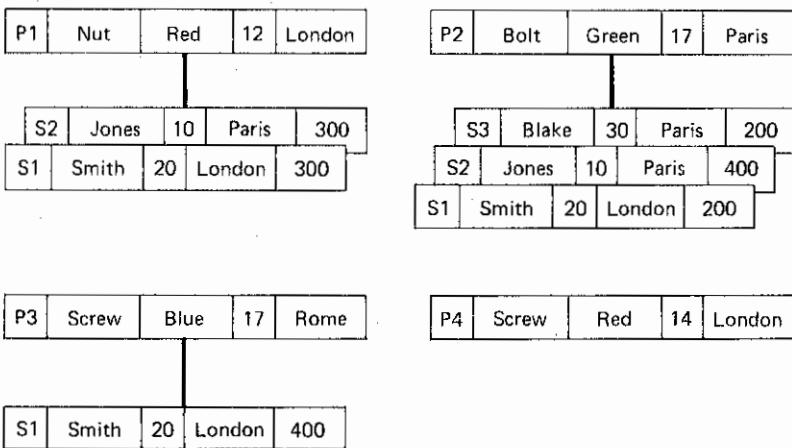


Fig. 3.3 Exemplo de dados em forma hierárquica (peças acima de fornecedores).

correspondente de embarque. Note que todo o conjunto de ocorrências de fornecedores de uma determinada ocorrência de peça pode conter qualquer quantidade de membros, inclusive zero (como no caso de P4).

O tipo de registro no topo da árvore – no nosso exemplo o tipo de registro peça – é usualmente conhecido como “raiz”. A Fig. 3.3 é um exemplo da estrutura hierárquica mais simples possível (não considerando o caso degenerado de uma hierarquia consistindo somente da raiz), com uma raiz e um só tipo de registro dependente. Em geral, a raiz pode ter qualquer quantidade de dependentes, cada um dos quais pode ter qualquer quantidade de dependentes de nível mais baixo, e assim sucessivamente, com qualquer número de níveis. Encontraremos exemplos de hierarquias mais complexas na Parte 3 deste livro.

Na seção anterior nós assemelhamos a visão relacional da Fig. 3.1 a três arquivos simples. Podemos semelhantemente assemelhar a visão hierárquica da Fig. 3.3 a um único arquivo, contendo registros organizados em três árvores separadas. A DML hierárquica discutida abaixo pode ser vista como uma coleção de operações sobre esses arquivos. Note, entretanto, que este arquivo é um objeto mais complexo do que as tabelas da Fig. 3.1. Em primeiro lugar, ele contém diversos tipos de registros, e não apenas um; no nosso exemplo são dois, um para peças e outro para fornecedores. Segundo, ele também contém *interligações* conectando ocorrências desses registros; no nosso exemplo há interligações entre ocorrências de peças e ocorrências de fornecedores, representando os embarques associados.

É fundamental para a visão hierárquica dos dados que qualquer ocorrência de registro só apresente seu significado completo quando visto no contexto – sem dúvida, nenhuma ocorrência dependente de registro pode sequer existir sem o seu superior. Portanto na DMI o análogo do relacional “obtenha o próximo no qual” tem que incluir um operando adicional (visto abaixo via uma cláusula “under” [sob]) para especificar este contexto, isto é, para identificar o superior da ocorrência objetivo (a não ser que a ocorrência obje-

tivo seja uma raiz). A Fig. 3.4 mostra um esboço da codificação requerida para manusear as duas consultas da Fig. 3.2 com a visão hierárquica da Fig. 3.3. Nós colocamos colchetes em volta de "next" (próximo) naquelas instruções em que esperamos que no máximo uma ocorrência satisfaça às condições especificadas. Também estamos supondo que "where" (no qual) possa ser omitido caso não desejemos especificar nenhuma condição particular a ser satisfeita.

Embora as consultas originais sejam simétricas, os dois procedimentos mostrados na Fig. 3.4 certamente não o são. (Contrasta com o caso relacional – Fig. 3.2 – onde a simetria original é mantida.) A perda de simetria é uma consequência direta da visão (Fig. 3.3) que é assimétrica em si mesma, com peças sendo tratadas como superiores e fornecedores como dependentes. Esta assimetria é um grande obstáculo da abordagem hierárquica, pois leva a complicações desnecessárias ao usuário. Especificamente, o usuário é forçado a dedicar tempo e esforço para resolver problemas que são introduzidos pela estrutura hierárquica de dados e que não são intrínsecas às perguntas feitas. É claro que isto piora rapidamente à medida que são introduzidos mais tipos de registros na estrutura e a hierarquia se torna mais complexa. Este não é um assunto trivial. Significa que os programas são mais complicados do que precisariam ser, com a consequência de que a escrita, depuração e manutenção irão requerer mais tempo de programadores do que deveriam.

Por outro lado, as hierarquias são um caminho óbvio e natural para modelar estruturas hierárquicas verdadeiras do mundo real. O exemplo de fornecedores e peças não está nesse caso, pois é uma correspondência de vários-para-vários entre fornecedores e peças. Departamentos e empregados fornecem um exemplo de estrutura hierárquica genuína (se for verdadeiro que cada empregado pertence a exatamente um departamento). Mas mesmo em estruturas hierárquicas genuínas surge o problema da assimetria na recuperação – considere, por exemplo, as consultas "Encontre os empregados de um determinado departamento" e "Encontre o departamento de um determinado empregado". Além disso, mesmo as estruturas hierárquicas genuínas tendem com o tempo a se tornarem estruturas vários-para-vários mais complexas. Nós retornaremos à questão de representar uma visão hierárquica de uma estrutura vários-para-vários na parte 3 deste livro.

No tocante a operações de atualização, encontramos que as estruturas hierárquicas como a da Fig. 3.3 apresentam características indesejáveis adicionais. Surgem anomalias ligadas a cada uma das três operações básicas (inserção, remoção, atualização). Mas, diferentemente dos problemas que discutimos anteriormente sobre recuperação, essas anom-

Q1: Encontre os números dos fornecedores que fornecem a peça P2	Q2: Encontre os números das peças fornecidas pelo fornecedor S2
<pre> get [next] part where P# = P2; do until no more parts; get next part; get [next] supplier under this part; print S#; end; </pre>	<pre> do until no more parts; get next part; get [next] supplier under this part where S# = S2; if found then print P#; end; </pre>

Fig. 3.4 Dois exemplos de consultas à visão hierárquica.

lias são diretamente devidas ao fato de estarmos lidando com uma situação de vários-para-vários; elas não surgiriam em uma situação de um-para-vários. As dificuldades estão ilustradas pelos três problemas simples retirados do final da seção anterior.

Inserção — Não é possível, sem a introdução de uma peça fictícia, inserir dados referentes a um novo fornecedor — digamos S4 — antes que aquele fornecedor forneça alguma peça.

Remoção — Uma vez que a informação de embarque está incorporada no tipo de registro de fornecedor, a única maneira de remover um embarque é remover a ocorrência correspondente de fornecedor. Segue-se então que se removermos um único embarque de um determinado fornecedor, perdemos toda a informação sobre aquele fornecedor. (As anomalias de inserção e remoção são na realidade duas faces de uma mesma moeda.) Por exemplo, a remoção do embarque que conecta P2 a S3 é manuseada pela remoção da ocorrência de S3 sob a peça P2, que — sendo a única ocorrência de S3 — causa a perda de toda a informação sobre S3.

Incidentalmente, um problema semelhante ocorre quando desejamos remover uma peça que seja a única fornecida por um certo fornecedor, pois a remoção de qualquer ocorrência de registro automaticamente remove também todas as ocorrências dependentes, em linha com a filosofia hierárquica.

Atualização — Se necessitarmos modificar a descrição de um fornecedor — por exemplo, mudar a cidade do fornecedor S1 para Amsterdam —, enfrentaremos ou o problema de pesquisar toda a visão para encontrar cada ocorrência do fornecedor S1, ou a possibilidade de introduzir uma inconsistência (o fornecedor S1 poderia aparecer como estando em Amsterdam em um ponto e em Londres em outro).²

3.4 A ABORDAGEM DE REDE

A Fig. 3.5 mostra uma *visão em rede* do banco de dados de fornecedores e peças. Nesta visão, como na abordagem hierárquica, os dados estão representados por registros e interligações. Entretanto, a rede é uma estrutura mais geral do que a hierárquica porque uma determinada ocorrência de registro pode ter *qualquer quantidade* de superiores imediatos (bem como qualquer quantidade de dependentes imediatos) — não estamos limitados ao máximo de um como no caso hierárquico. Por isso a abordagem em rede nos permite modelar uma correspondência de vários-para-vários mais diretamente do que a abordagem hierárquica, como mostra a Fig. 3.5. Em adição aos tipos de registros que representam os fornecedores e as peças, nós introduzimos um terceiro tipo de registro, que chamaremos de conector. Uma ocorrência de conector representa a associação (embarque) entre um fornecedor e uma peça, e contém dados que descrevem essa associação (no exemplo, a quantidade de peças fornecidas). Todas as ocorrências de conector de um determinado fornecedor estão colocadas em uma cadeia³ que começa e termina naquele fornecedor. De forma semelhante, todas as ocorrências de conector de uma determinada peça são co-

² Este problema é consequência da redundância na estrutura de dados. Entretanto, se a redundância for controlada (veja o Capítulo 1), o problema será do sistema e não do usuário.

³ Estas cadeias podem ser representadas fisicamente na memória por cadeias reais de indicadores de localização ou por algum método funcionalmente equivalente. No entanto, o usuário sempre poderá *imaginar* que as cadeias existam fisicamente, não importando a implementação realmente feita.

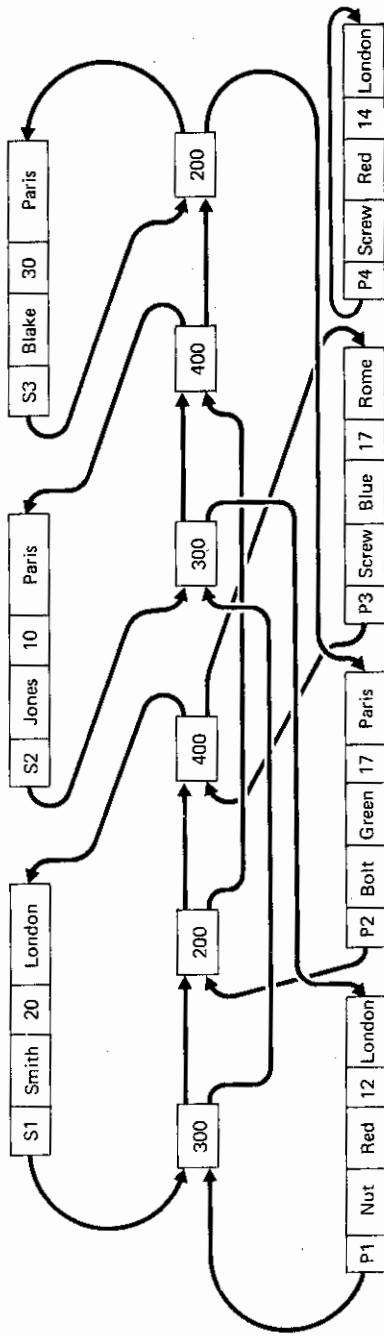


Fig. 3.5 Exemplo de dados em forma de rede.

locadas em uma cadeia que começa e termina naquela peça. Cada ocorrência de conector encontra-se portanto em exatamente duas cadeias, uma de fornecedor e uma de peça. Por exemplo, a Fig. 3.5 mostra que o fornecedor S2 fornece 300 peças P1 e 400 peças P2; igualmente ela mostra que a peça P1 é fornecida na quantidade de 300 pelo fornecedor S1 e na quantidade de 300 pelo fornecedor S2. Note, incidentalmente, que a correspondência entre, digamos, um fornecedor e os registros conectores associados é de um-para-vários, o que mostra que hierarquias podem ser facilmente representadas em um sistema em rede.

Novamente nós podemos assemelhar a visão a um arquivo de registros e interligações; a estrutura interna deste arquivo é mais complexa do que no caso hierárquico. Como a Fig. 3.6 ilustra, nós precisamos agora na DML não somente do operador "traga o próximo no qual", mas também de um "traga o acima" (*get over*) para buscar a ocorrência superior única específica de conector em uma cadeia específica. (Na verdade o operador "traga o acima" também é necessário para hierarquias, mas neste caso não é preciso especificar que cadeia seguir.)

Q1: Encontre os números dos fornecedores que fornecem a peça P2	Q2: Encontre os números das peças fornecidas pelo fornecedor S2
<pre> get [next] part where P# = P2; do until no more connectors under this part; get next connector under this part; get supplier over this connector; print S#; end; </pre>	<pre> get [next] supplier where S# = S2; do until no more connectors under this supplier; get next connector under this supplier; get part over this connector; print P#; end; </pre>

Fig. 3.6 Dois exemplos de consultas à visão em rede.

A estrutura em rede da Fig. 3.5 é mais simétrica do que a estrutura hierárquica da Fig. 3.3; e a simetria está refletida nos dois procedimentos da Fig. 3.6. No entanto, estes procedimentos são significativamente mais complicados do que tanto (a) seus-análogos relacionais (Fig. 3.2), por um lado, quanto (b) pelo menos a solução hierárquica para a consulta Q1, por outro lado (Fig. 3.4). Portanto simetria não é tudo.

Outra complicação, que não está ilustrada na Fig. 3.6, ocorre em conexão com consultas tipo "Encontre a quantidade de peças P2 fornecidas pelo fornecedor S2". Para responder a esta consulta temos que ler a ocorrência (única) de conector que existe tanto na cadeia de S2 como na cadeia de P2. O problema é que existem duas estratégias para se localizar esta ocorrência, uma que começa no fornecedor e pesquisa sua cadeia para encontrar um conector ligado à peça, e outra que começa na peça e pesquisa sua cadeia procurando um conector ligado ao fornecedor. Como o usuário irá decidir que estratégia adotar? A escolha pode ser significativa.

Observações semelhantes aplicam-se às operações de atualização. Encontramos que as anomalias discutidas para hierarquias na Seção 3.3 não surgem na rede da Fig. 3.5.⁴ No entanto, a programação envolvida não é sempre tão imediata como possa parecer. Por exemplo, no caso de remoção abaixo nós vamos encontrar o problema de estratégia descrito no parágrafo anterior.

Inserção — Para inserir dados referentes a um novo fornecedor — digamos S4 — nós simplesmente criamos uma nova ocorrência de registro de fornecedor. Inicialmente não haverá nenhum registro conector para este novo fornecedor; sua cadeia consiste de um único indicador de localização do fornecedor para ele mesmo.

Remoção — Para remover o embarque conectando P2 e S3 nós removemos a ocorrência de registro conector que liga este fornecedor e esta peça. As duas cadeias envolvidas terão que ser apropriadamente ajustadas (provavelmente esses ajustamentos serão feitos de forma automática).

Atualização — Podemos mudar a cidade do fornecedor S1 para Amsterdam sem problemas de pesquisa e sem a possibilidade de inconsistências, pois a cidade de S1 aparece em um local preciso na estrutura.

Nós sustentamos, portanto, que a maior desvantagem da abordagem em rede é a excessiva complexidade, tanto na estrutura de dados em si como na DML associada. A origem da complexidade reside na faixa de construções de transporte de informações suportadas na estrutura em rede (duas foram ilustradas neste capítulo, registros e interligações, mas os sistemas em rede tipicamente suportam também outras). Em geral, quanto mais construções existem, mais operadores são necessários para manuseá-las, e consequentemente mais complicada se torna a DML. Nós vamos discutir esses itens mais longamente adiante neste livro.

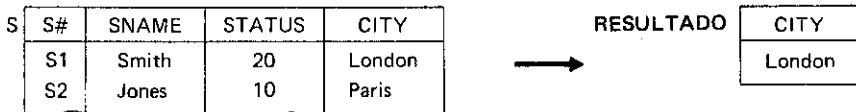
3.5 OPERADORES DE NÍVEL MAIS ALTO

Até agora em nossas discussões sobre linguagens de manipulação de dados nós supusemos tacitamente que, como na programação de arquivos convencionais, todos os operadores lidariam essencialmente com um registro de cada vez. No entanto, muitos problemas são expressos mais naturalmente não em termos de registros individuais, mas sim em termos de conjuntos (considere as consultas Q1 e Q2 da Fig. 3.2, por exemplo). Nesta seção introduziremos a possibilidade de linguagens mais poderosas — linguagens nas quais os operadores são capazes de manipular conjuntos inteiros de objetos simples, sem a restrição de um registro de cada vez.

Vamos primeiramente considerar a operação de recuperação e começar observando algumas consultas de exemplo sobre a estrutura relacional da Fig. 3.1.

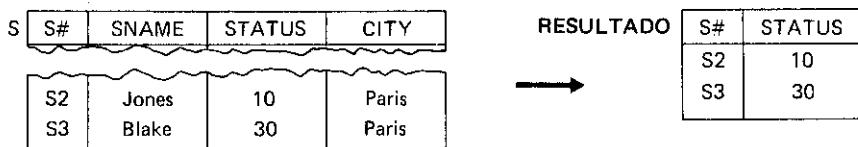
⁴ As dificuldades não desaparecem simplesmente devido à abordagem em rede por si, mas sim em função do formato particular que a rede assume. Uma qualificação semelhante se aplica à nossa discussão sobre operações de atualização na abordagem relacional (Seção 3.2). O problema real é de projeto e normalização do banco de dados, detalhes que estão além do escopo do presente capítulo; veja o capítulo 14.

3.5.1 Encontre a cidade (CITY) do fornecedor S1.



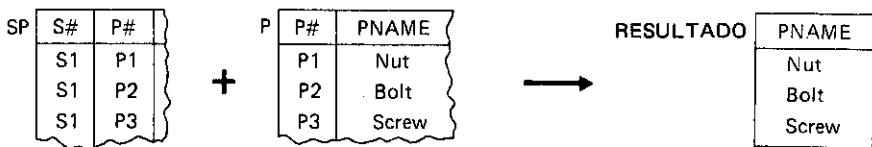
A resposta é “London”. Para ser mais específico, a resposta é uma tabela (relação) com uma linha e uma coluna, sendo esta coluna baseada no domínio das localizações (cidades) e contendo um único valor “London”.

3.5.2 Encontre S# e STATUS para os fornecedores de Paris.



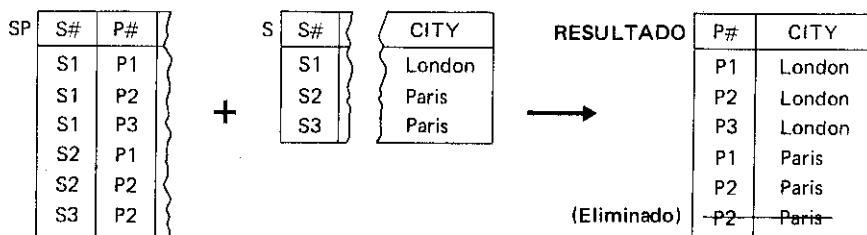
O resultado é novamente uma tabela, desta vez com duas linhas e duas colunas.

3.5.3 Encontre PNAME das peças fornecidas pelo fornecedor S1.



Novamente o resultado é uma tabela. De fato, o resultado de *qualquer* operação de recuperação pode ser considerado como uma tabela, e este aspecto é de considerável significado como veremos mais tarde (no Capítulo 12). Neste exemplo particular, a tabela resultante é um subconjunto de uma tabela única, como era nos dois exemplos precedentes, mas as *duas* tabelas têm que ser examinadas na construção deste resultado.

3.5.4 Para cada peça fornecida, encontre P# e os nomes de todas as cidades que fornecem a peça.



Neste exemplo final, não somente é necessário novamente examinar duas tabelas, mas também os valores do resultado derivam realmente de duas tabelas. Note, incidentalmente, que uma linha duplicata redundante é eliminada do resultado final; a razão para isto é que, matematicamente falando, uma tabela (relação) é um conjunto — um conjunto de linhas — e conjuntos por definição não podem conter elementos em duplicata. Nós vamos discutir este item com mais detalhes no Capítulo 4.

Em geral, portanto, o resultado de qualquer recuperação é uma tabela, derivada de alguma forma de outras tabelas no banco de dados; pode ser envolvida qualquer quantidade de tabelas na formação do resultado, tanto no condicionamento da seleção como no fornecimento de valores para o resultado. Em outras palavras, *o processo de recuperação é, precisamente, um processo de construção de tabela*. Reconhecido este fato, nós podemos definir um conjunto de *operadores de construção de tabelas* para uso na recuperação. Vamos discutir brevemente três desses operadores: SELECT, PROJECT, e JOIN.

O operador SELECT constrói uma nova tabela usando um *subconjunto horizontal* de uma tabela existente, isto é, todas as linhas de uma tabela existente que satisfaçam a certa condição. O operador PROJECT, em contraste, forma um *subconjunto vertical* de uma tabela existente, extraíndo colunas específicas e removendo qualquer linha duplicata redundante no conjunto de colunas extraídas. Usando esses dois operadores nós podemos imediatamente escrever programas para os primeiros dois exemplos acima.

3.5.1 Encontre CITY para o fornecedor S1.

Etapa 1. `SELECT S WHERE S#='S1' GIVING TEMP`

Esta etapa nos fornece a seguinte tabela (basicamente uma cópia da linha “fornecedor S1” da tabela S).

TEMP	S#	SNAME	STATUS	CITY
	S1	Smith	20	London

Etapa 2. `PROJECT TEMP OVER CITY GIVING RESULT`

Esta etapa extrai (isto é, copia) a coluna CITY de TEMP, fornecendo o resultado desejado.

3.5.2 Encontre S# e STATUS para os fornecedores de Paris.

```
SELECT S WHERE CITY='PARIS' GIVING TEMP  
PROJECT TEMP OVER (S#, STATUS) GIVING RESULT
```

Este exemplo é muito semelhante ao exemplo 3.5.1.

Os dois exemplos restantes — os que envolvem duas tabelas — requerem o uso do operador JOIN. Se duas tabelas possuem cada uma coluna definida em um domínio comum, elas podem ser *joined* (unidas) com base nessas duas colunas; o resultado da junção é uma tabela nova e mais ampla, na qual cada linha é formada pela concatenação de duas linhas, uma de cada tabela original, sendo que as duas linhas têm o mesmo valor naquelas duas colunas. Por exemplo, as tabelas S e P podem ser unidas com base em suas colunas CITY; o resultado está mostrado na Fig. 3.7.

Nós mudamos os nomes das duas colunas CITY para SCITY e PCITY para evitar ambiguidade. Note que se uma linha de uma das tabelas originais não tiver correspondente

na outra, ela simplesmente não participa do resultado; por exemplo, P3 (armazenado em Roma) não aparece na junção na Fig. 3.7.

[A tabela na Fig. 3.7 contém duas colunas idênticas. Este fato segue-se necessariamente da definição que demos para a operação de junção. Na verdade, aquela definição corresponde apenas a uma das várias uniões possíveis — ou seja, a junção na qual a “condição de junção” está baseada na *igualdade* entre valores na coluna comum. Esta junção é por isso conhecida como *equi-join*. É também possível definir, por exemplo, uma junção “maior do que”, uma junção “não igual”, etc. — embora a equijoin seja de longe a mais freqüentemente usada. Somente a equijoin contém necessariamente duas colunas idênticas. Naturalmente sempre é possível eliminar-se uma daquelas duas colunas por meio da operação PROJECT; uma equijoin com uma das colunas idênticas eliminada é chamada de *junção natural*. A junção natural é importante no contexto de *normalização adicional* (a ser discutida no Capítulo 14). No momento, entretanto, nós vamos usar o termo “junção” para significar a equijoin.]

S#	SNAME	STATUS	SCITY	P#	PNAME	COLOR	WEIGHT	PCITY
S1	Smith	20	London	P1	Nut	Red	12	London
S1	Smith	20	London	P4	Screw	Red	14	London
S2	Jones	10	Paris	P2	Bolt	Green	17	Paris
S3	Blake	30	Paris	P2	Bolt	Green	17	Paris

Fig. 3.7 Junção de S e P em CITY.

Agora nós podemos programar os outros dois exemplos.

3.5.3 Encontre PNAME das peças fornecidas pelo fornecedor S1.

```
SELECT SP WHERE S#='S1' GIVING TEMP1
JOIN TEMP1 AND P OVER P# GIVING TEMP2
PROJECT TEMP2 OVER PNAME GIVING RESULT
```

3.5.4 Para cada peça fornecida, encontre P# e os nomes de todas as cidades que fornecem a peça.

```
JOIN SP AND S OVER S# GIVING TEMP
PROJECT TEMP OVER (P#, CITY) GIVING RESULT
```

Neste último exemplo, a definição de PROJECT garante que não aparecerão linhas duplicadas no resultado.

Os operadores SELECT, PROJECT, e JOIN, juntamente com outros que serão discutidos no Capítulo 12, constituem unidos a *álgebra relacional*. Cada operação da álgebra relacional usa uma ou duas relações como seu(s) operando(s) e produz como resultado uma nova relação. Como nós já ilustramos, é claramente possível prover o usuário com uma linguagem de manipulação de dados na qual esses operadores estejam diretamente disponíveis; portanto uma DML de nível mais alto (manuseando conjuntos) é perfeitamente exequível, pelo menos no que tange à recuperação. Em relação a outras operações, vamos nos contentar por enquanto com a informação de que é sem dúvida possível definir

operadores de inserção, remoção e atualização que, como os operadores da álgebra relacional, lidem com conjuntos inteiros como simples operandos. Exemplos desses operadores serão mostrados na Parte 2 deste livro. Portanto, pelo menos para sistemas relacionais, é definitivamente executável uma DML a nível de conjuntos, e muitos sistemas relacionais já fornecem uma linguagem desse nível. Sem dúvida, um dos pontos fortes da abordagem relacional é que linguagens tais como a álgebra relacional, que são simples e ainda assim muito poderosas, podem ser definidas tão diretamente.

E sobre as abordagens hierárquica e em rede? Seria errôneo inferir que não podem ser definidas linguagens a nível de conjuntos para esses sistemas. Entretanto, novamente o fato de haver mais de uma forma para se representar a informação na estrutura de dados conduz à necessidade de existir mais do que um conjunto de operadores na DML. Esta afirmativa é verdadeira independentemente do nível da linguagem. Portanto, sem entrar em detalhes, nós afirmamos que uma linguagem a nível de conjunto hierárquica ou em rede é necessariamente mais complexa do que uma linguagem relacional a nível de conjunto. Mais uma vez, nós iremos discutir esta questão em mais detalhes adiante neste livro.

Nesta seção, nós nos concentraremos na álgebra relacional. Diversos sistemas relacionais fornecem uma DML que é diretamente baseada nesta álgebra. Entretanto, desde que a álgebra foi inicialmente desenvolvida, várias outras linguagens foram projetadas para operarem sobre relações, todas pelo menos tão poderosas quanto a álgebra, e muitas delas ainda mais fáceis de serem usadas. Entre estas estão ALPHA e QUEL, ambas baseadas em *cálculo relacional*; SQUARE e SEQUEL (depois rebatizada SQL), baseadas em uma operação conhecida como “mapeamento”; e pelo menos duas linguagens gráficas, Query By Example e CUPID, primariamente orientadas para uso em terminais com telas. Nós vamos descrever algumas dessas linguagens em capítulos subsequentes.

3.6 RESUMO

Na Seção 1.2 nós afirmamos que um sistema de banco de dados tem que ser capaz de representar dois tipos de objetos, “entidades” e “relacionamentos”. Nós também afirmamos que fundamentalmente não existe uma diferença real entre os dois; um relacionamento é meramente um caso especial de entidade. As três abordagens (relacional, hierárquica e em rede) diferem na forma como elas permitem ao usuário ver e manipular *relacionamentos*.

Na abordagem relacional os relacionamentos são representados da mesma maneira que outras entidades, por exemplo como tuplas em relações. Nas abordagens hierárquica e em rede certos relacionamentos⁵ são representados por meio de “interligações”. Basicamente essas interligações são capazes de representar associações um-para-vários; a diferença entre as abordagens em rede e hierárquica é que na primeira as interligações podem ser combinadas para modelar associações mais complexas vários-para-vários, o que não é possível com a última. Outra diferença, que não foi enfatizada no presente capítulo, é que as interligações geralmente recebem nomes em uma rede e são anônimas em uma hierarquia, por razões que ultrapassam o escopo deste livro.

5

Nem todos, entretanto. Na estrutura hierárquica da Fig. 3.3, por exemplo, o relacionamento “embarques que possuem certos números de peças” está representado por interligações, enquanto que o relacionamento “fornecedores que têm determinada localização” está representado pela igualdade dos valores de CITY nas ocorrências de registros pertinentes. O último método de representar relacionamentos é o único suportado na abordagem relacional. Nós vamos retornar a este tópico no capítulo 28.

Concluímos este capítulo identificando alguns sistemas que podem ser considerados como representativos das três abordagens. Alguns dos sistemas que existem há mais tempo são *hierárquicos*; como exemplos nós podemos apontar o Information Management System, IMS, da IBM; o Mark IV, da Informatics; o System 2000, da MRI; e o precursor mais recente, o Time-Shared Data Management System, TDMS, da SDC. O IMS fornece uma DML conhecida como DL/I (Data Language/I) de um registro por vez, com a qual nós lidaremos em mais detalhes na Parte 3 deste livro. O sistema 2000 é uma linguagem orientada para conjunto, que, entretanto, não possui a generalidade total da álgebra relacional introduzida na Seção 3.5.

O exemplo mais importante de um sistema *em rede* é fornecido pelas especificações do CODASYL Data Base Task Group (DBTG) e seus vários comitês sucessores. Diversos sistemas comercialmente disponíveis estão baseados nessas propostas, entre eles o DMS 1100 da UNIVAC e o IDMS da Cullinane. Outros sistemas em rede incluem o TOTAL, da Cincom, o DBOMP, da IBM, e o Integrated Data Store, IDS, da Honeywell, do qual foram derivadas várias idéias do DBTG. Nós vamos discutir o DBTG em profundidade na parte 4 deste livro.

Como sistemas *relacionais*, nós podemos identificar o MAGNUM, da Tymshare, e o Query By Example da IBM (já mencionado), como sistemas comercialmente disponíveis. O MAGNUM possui uma linguagem de um registro por vez, incluindo amplos recursos computacionais e de relatórios, que podem ser usados tanto sob a forma de comandos a partir de um terminal *on-line* como na forma mais tradicional de programação em lotes (*batch*). O Query By Example será tratado em detalhes no Capítulo 11. Devemos mencionar também o sistema NOMAD da NCSS, que suporta tanto hierarquias como relações, e inclui diversos operadores — em particular, diversas formas de união — entre seus dispositivos de geração de relatórios. Em adição a esses sistemas comerciais (comparativamente recentes), um grande número de sistemas experimentais foi e continua sendo desenvolvido em universidades e instituições similares. Mencionaremos, em particular, o System R, da IBM Research, e o INGRES, da Universidade da Califórnia, em Berkeley. O sistema R está descrito em profundidade na Parte 2 deste livro, e o INGRES está também mencionado em muitos pontos, particularmente no Capítulo 13. Esses dois sistemas fornecem linguagens de manipulação de conjuntos que realmente são *mais* poderosas do que a álgebra relacional.

EXERCÍCIOS

Um banco de dados contém informações sobre pessoas e suas habilidades. Em um certo momento, as seguintes pessoas estão representadas no banco de dados, juntamente com suas habilidades, como indicado abaixo.

Pessoa	Habilidade
Arthur	Programação
Bill	Operação e Programação
Charlie	Engenharia, Programação e Operação
Dave	Operação e Engenharia

Para cada uma destas pessoas, o banco de dados contém vários detalhes pessoais, tal como endereço. Para cada habilidade, ele contém uma identificação dos cursos básicos de treinamento apropriados, um código associado ao nível de trabalho, e outras informações. O banco de dados também contém a data em que cada pessoa participou de cada curso, onde isto for aplicável (parte-se do princípio de que a participação no curso é essencial antes que se possa considerar a habilidade obtida).

3.1 Esboce a estrutura relacional para estes dados.

- 3.2 Esboce *duas* estruturas hierárquicas para estes dados.
- 3.3 Esboce uma estrutura em rede para estes dados.
- 3.4 Para cada uma das suas respostas às três primeiras questões, forneça um procedimento básico para se encontrar os nomes de todas as pessoas (a) que possuem determinada habilidade; (b) possuindo pelo menos uma habilidade em comum com uma pessoa especificada. No caso relacional, forneça soluções usando os dois níveis de linguagem (um registro por vez, um conjunto por vez) introduzidos neste capítulo.

REFERÊNCIAS E BIBLIOGRAFIA

Além das referências listadas abaixo, chamo a atenção do leitor para a ACM Computing Surveys 8, nº 1 (edição especial sobre sistemas de gerenciamento de bancos de dados), que inclui estudos sobre cada uma das três abordagens.

3.1 R. E. Bleier. "Treating Hierarchical Data Structures in the SDC Time-Shared Data Management Systems (TDMS)", *Proc. ACM National Meeting* (1967).

3.2 R. E. Bleier and A. H. Vorhaus. "File Organization in the SDC Time-Shared Data Management System (TDMS)." *Proc. IFIP Congress* (1968).

Descreve a estrutura de armazenamento TDMS (uma organização invertida).

3.3 C. W. Bachman and S. B. Williams. "A General Purpose Programming System for Random Access Memories." *Proc. FJCC*, AFIPS Press (1964).

Uma das descrições mais antigas do IDS (precursor do DBTG). Bachman foi o arquiteto original do IDS.

Parte 2

A Abordagem Relacional

Os conceitos básicos da abordagem relacional foram introduzidos no Capítulo 3. A Parte 2 consiste de um tratamento mais detalhado sobre aquelas idéias. O Capítulo 4 lida mais extensamente com estruturas de dados relacionais, discutindo noções fundamentais como domínio, chave e forma normalizada com alguma profundidade. Os Capítulos 5-10 examinam um sistema específico, o Sistema R, com muitos detalhes. Esses capítulos servem não somente como uma introdução abrangente ao Sistema R especificamente, mas também como uma ilustração dos vários aspectos dos sistemas relacionais em geral. O Capítulo 11 descreve um outro sistema, o Query By Example, com um pouco menos de detalhes. O Capítulo 12 apresenta um tratamento expandido da álgebra relacional introduzida no Capítulo 3, e enfatiza a natureza fundamental da álgebra como um componente do *modelo relacional*. O Capítulo 13 discute uma abordagem alternativa ao projeto de linguagens relacionais, baseada no *cálculo relacional*. Finalmente, o Capítulo 14 (Normalização Adicional) concentra-se no problema da escolha do conjunto de relações mais apropriado para representar uma determinada coleção de dados, isto é, o problema do projeto do banco de dados relacional.

4

Estrutura Relacional de Dados

4.1 RELAÇÕES

Definição — Dada uma coleção de conjuntos D_1, D_2, \dots, D_n (não necessariamente distintos), R é uma *relação* naqueles n conjuntos se ele for um conjunto de n -tuplas ordenadas $\langle d_1, d_2, \dots, d_n \rangle$ tais que d_1 pertença a D_1 , d_2 pertença a D_2, \dots, d_n pertença a D_n . Os conjuntos D_1, D_2, \dots, D_n são os *domínios* de R. O valor n é o *grau* de R.

A Fig. 4.1 ilustra uma relação denominada PART, de grau 5. Os cinco domínios são conjuntos de valores representando, respectivamente, números de peças, nomes de peças, pesos de peças, cores de peças e as localizações onde as peças estão armazenadas. O domínio “part color” (cor da peça), por exemplo, é um conjunto de todas as cores válidas de peças; note que podem existir cores incluídas neste domínio que não aparecem na relação PART neste momento específico.

Como a figura ilustra, é conveniente representar-se uma relação como uma tabela. (A tabela 4.1 é, na realidade, uma versão estendida da tabela P na Fig. 3.1.) Cada linha da tabela representa uma tupla- n (ou simplesmente uma tupla) da relação. O número de tu-

PART	P#	PNAME	COLOR	WEIGHT	CITY
	P1	Nut	Red	12	London
	P2	Bolt	Green	17	Paris
	P3	Screw	Blue	17	Rome
	P4	Screw	Red	14	London
	P5	Cam	Blue	12	Paris
	P6	Cog	Red	19	London

Fig. 4.1 A relação PART.

plas em uma relação é chamado de *cardinalidade* da relação; isto é, a cardinalidade da relação PART é seis.

Relações de grau um são chamadas de *unárias*; como exemplos, veja as relações resultado dos exemplos 3.5.1 e 3.5.3 (Seção 3.5). Semelhantemente, relações de grau dois são *binárias* (como exemplos, veja as relações resultado nos exemplos 3.5.2 e 3.5.4), relações de grau três são *ternárias*, ..., e relações de grau n são n -árias.

Temos outra definição equivalente de relação, que é algumas vezes útil. Primeiramente vamos definir a noção de *produto Cartesiano*. Dada uma coleção de conjuntos D_1, D_2, \dots, D_n (não necessariamente distintos), o produto cartesiano desses n conjuntos, escrito $D_1 \times D_2 \times \dots \times D_n$, é o conjunto de todas as possíveis n -tuplas ordenadas $\langle d_1, d_2, \dots, d_n \rangle$ tal que d_1 pertença a D_1 , d_2 pertença a D_2 , ..., d_n pertença a D_n . Por exemplo, a Fig. 4.2 mostra o produto Cartesiano de dois conjuntos S# e P#.

Agora definamos R como sendo uma relação dos conjuntos D_1, D_2, \dots, D_n se ele for um subconjunto do produto Cartesiano $D_1 \times D_2 \times \dots \times D_n$.

Estritamente falando, não há ordenação definida entre as tuplas de uma relação, pois uma relação é um conjunto e conjuntos não são ordenados. Na Fig. 4.1, por exemplo, as tuplas da relação PART poderiam perfeitamente ter sido mostradas na seqüência inversa — e continuaria sendo a mesma relação. Entretanto, há situações em que se torna muito conveniente a possibilidade de se garantir alguma ordenação específica, de tal forma que nós saibamos, por exemplo, que o operador “traga o próximo” lerá tuplas de PART em ordem ascendente de número de peça. Em um contexto de banco de dados, por isso, nós freqüentemente consideramos que há uma ordenação — de fato temos que agir assim, para que o “traga o próximo” tenha significado — mas *das duas uma*: (a) ou a ordenação é definida pelo sistema, isto é, o usuário não se preocupa com qual ela seja (enquanto ela for estável), *ou* (b) ela é definida em termos dos valores que aparecem dentro da relação; por exemplo, seqüência ascendente por número de peça (*ordenação controlada pelo valor*). Qualquer outro tipo de ordenação, como por exemplo primeiro a chegar/primeiro a sair, ou controlada por programa, está especificamente excluído. Nós vamos retornar a este assunto no Capítulo 28.

Voltando à definição original, nós podemos ver que, em contraste, os domínios de uma relação têm uma ordenação definida entre eles (uma relação é um conjunto de n -tuplas *ordenadas*, sendo o j -ésimo elemento de cada n -tupla retirado do j -ésimo domínio). Se rearranjássemos as cinco colunas da relação PART (Fig. 4.1) em outra ordenação dife-

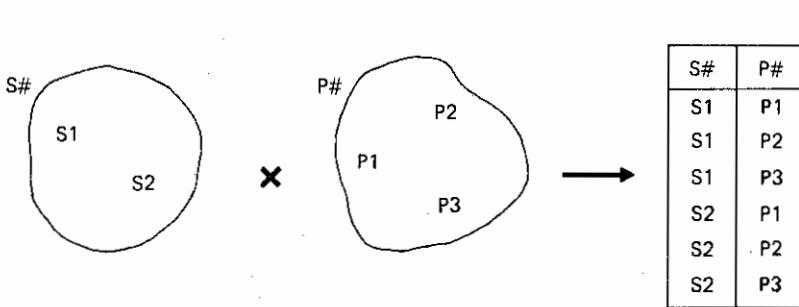


Fig. 4.2 Exemplo de um produto cartesiano.

rente, a tabela resultante seria uma relação diferente, matematicamente falando. Entretanto, como os usuários normalmente se referem às colunas por seus nomes e não por sua posição relativa, muitos sistemas minimizam esta restrição e tratam a ordem das colunas como se fosse tão irrelevante como a ordem das linhas. Neste livro, nós vamos geralmente considerar que a ordenação de colunas é insignificante, a menos que seja explicitamente estabelecido o contrário.

4.2 DOMÍNIOS E ATRIBUTOS

É importante observar a diferença entre um domínio, por um lado, e colunas – ou *atributos* – que são retiradas daquele domínio, pelo outro. Um atributo representa o *uso* de um domínio dentro de uma relação. Para enfatizar a distinção, nós podemos dar nomes aos atributos distintos dos domínios básicos; por exemplo, veja a Fig. 4.3.

Nesta figura nós temos parte de um esquema relacional, no qual foram declarados cinco domínios (PART_NUMBER, PART_NAME etc.) e uma relação (PART). A relação está definida com cinco atributos (P#, PNAME etc.), e cada atributo está especificado como tendo sido retirado de um domínio correspondente. O esquema está escrito em uma linguagem hipotética de definição de dados.

Nós freqüentemente usaremos uma convenção que nos permita omitir a especificação do domínio da declaração de um atributo, se o atributo ostentar o mesmo nome. Entretanto nem sempre é possível fazer-se isto, como mostra o exemplo da Fig. 4.4.

Neste exemplo, nós temos uma relação com três atributos mas somente dois domínios distintos. (Veja na definição original, na Seção 4.1, que os domínios “não têm que ser necessariamente distintos”.) O significado de uma tupla na relação COMPONENT é que as peças maiores incluem as peças menores, na quantidade indicada, como componentes imediatos. Os dois domínios distintos são P# (nímeros de peças) e QUANTITY. O exemplo ilustra outra convenção comum, o de se gerar nomes distintos de atributos prefixando-se o nome do domínio com *nomes funcionais* para indicar as diferentes funções assumidas por aquele domínio em cada uma das suas participações.

Vamos agora introduzir a idéia de *normalização*. Todas as relações em um banco de dados relacional têm que satisfazer à seguinte condição.

```
DOMAIN PART_NUMBER CHARACTER (6)
DOMAIN PART_NAME  CHARACTER (20)
DOMAIN COLOR      CHARACTER (6)
DOMAIN WEIGHT     NUMERIC   (4)
DOMAIN LOCATION   CHARACTER (15)

RELATION PART
  (P#      : DOMAIN PART_NUMBER,
   PNAME   : DOMAIN PART_NAME,
   COLOR   : DOMAIN COLOR,
   WEIGHT  : DOMAIN WEIGHT,
   CITY    : DOMAIN LOCATION)
```

Fig. 4.3 Domínios versus atributos.

COMPONENT	MAJOR_P#	MINOR_P#	QUANTITY
	P1	P2	2
	P1	P4	4
	P5	P3	1
	P3	P6	3
	P6	P1	9
	P5	P6	8
	P2	P4	3

Fig. 4.4 A relação COMPONENT.

- Cada valor na relação — isto é, cada valor de atributo em cada tupla — é *atômico* (isto é, não é decomponível no tocante ao sistema).

Colocando de outra maneira, para cada posição linha/coluna na tabela existirá precisamente um valor, nunca um conjunto de valores. (São permitidos valores *nulos* — isto é, valores especiais representando “desconhecido” ou “inaplicável”, como no caso de “horas trabalhadas” para um empregado em férias.) Uma relação que satisfaça à condição acima é dita ser *normalizada*.¹

É trivial passar uma relação não-normalizada para uma forma normalizada equivalente. Vamos dar um exemplo simples aqui; para discussões adicionais sobre esse processo veja o Capítulo 14, especialmente a Seção 14.6 e o Exercício 14.1. A relação BEFORE (veja figura 4.5) está definida nos domínios S# (números dos fornecedores) e PQ (quantidade de peças); os elementos de PQ são em si mesmos relações definidas nos domínios P# (números de peças) e QTY (quantidade), e portanto BEFORE está não-normalizada. A relação AFTER é uma relação equivalente normalizada. (O significado de cada uma destas relações é que os fornecedores indicados fornecem as peças indicadas nas quantidades indicadas.)

Matematicamente falando, BEFORE é uma relação, de grau dois, mas é uma relação na qual nem todos os domínios básicos são *simples*. (Um domínio simples é aquele em que todos os elementos são atômicos.) AFTER é uma relação de grau três semanticamente equivalente, com a propriedade de que seus domínios são simples — em outras palavras, AFTER está normalizada. Nós escolhemos dar suporte somente às relações normalizadas na abordagem relacional porque (a) como o exemplo mostra, esta escolha não traz nenhuma restrição real ao que pode ser representado, e (b) a simplificação resultante na estrutura de dados leva a simplificações correspondentes em numerosas outras áreas — particularmente nos operadores da DML. Daqui por diante nós partiremos do princípio de que as relações estarão sempre normalizadas.

¹ Equivalentemente, tal relação é dita estar na *primeira forma normal*. No Capítulo 14 estão descritas formas normais adicionais.

BEFORE	S#	PQ			AFTER
		P#	QTY		
S1	P1	300		S1	P1 300
	P2	200		S1	P2 200
	P3	400		S1	P3 400
	P4	200		S1	P4 200
	P5	100		S1	P5 100
	P6	100		S1	P6 100
S2	P1	300		S2	P1 300
	P2	400		S2	P2 400
S3	P2	200		S3	P2 200
	P2	200		S4	P2 200
S4	P2	200		S4	P4 300
	P4	300		S4	P5 400
	P5	400			

Fig. 4.5 Um exemplo de normalização.

4.3 CHAVES

É freqüente o caso em que dentro de uma dada relação haja um atributo cujos valores sejam únicos naquela relação e que portanto possam ser usados para identificar as tuplas daquela relação. Por exemplo, o atributo P# da relação PART tem esta propriedade — cada tupla de PART contém um valor distinto de P#, e este valor pode ser usado para distinguir esta tupla de todas as outras na relação. P# é dito ser a *chave primária* de PART.

Nem toda relação possui uma chave primária de atributo único. Entretanto, toda relação *possui* alguma forma de combinação de atributos que, quando unidos, ficam com a propriedade de uma identificação única; uma “combinacão” que consista de um único atributo constitui um mero caso especial. Na relação AFTER da Fig. 4.5, por exemplo, a combinação (S#, P#) tem esta propriedade; o mesmo ocorre com a combinação (MAJOR_P#, MINOR_P#) na relação COMPONENT na Fig. 4.4. A existência de uma combinação como essas é garantida pelo fato de ser a relação um conjunto: Como conjuntos não têm elementos em duplicata, cada tupla de uma determinada relação é única no que tange àquela relação, e consequentemente obteremos a propriedade de identificação única, nem que seja através da combinação de *todos* os atributos. Na prática não costuma ser necessário envolver todos os atributos — é normalmente suficiente uma combinação menor. Por isso toda relação possui uma chave primária (possivelmente composta). Para nós, a chave primária é não-redundante, no sentido de que nenhum dos seus atributos constituintes é supérfluo para o objetivo da identificação única; por exemplo, a combinação (P#, COLOR) não é uma chave primária de PART.

Ocasionalmente nós podemos encontrar uma relação na qual haja mais do que uma combinação de atributos possuindo a propriedade de identificação única, e portanto mais de uma *chave candidata*. A Fig. 4.6 ilustra uma relação como essa (SUPPLIER). A situa-

SUPPLIER	S#	SNAME	STATUS	CITY
	S1	Smith	20	London
	S2	Jones	10	Paris
	S3	Blake	30	Paris
	S4	Clark	20	London
	S5	Adams	30	Athens

Fig. 4.6 A relação SUPPLIER.

ção neste caso é que cada fornecedor possui sempre um único número de fornecedor e um único nome de fornecedor. Podemos aqui escolher arbitrariamente uma das candidatas, digamos S#, como chave primária para a relação. Uma chave candidata que não seja a chave primária, tal como SNAME no exemplo, recebe o nome de *chave alternativa*.

Até agora nós consideramos a chave primária sob um ponto de vista puramente formal, isto é, puramente como uma identificação de tuplas em uma relação, sem nenhuma atenção a como essas tuplas serão interpretadas. Tipicamente, entretanto, essas tuplas representam *entidades* do mundo real, e a chave primária serve como um identificador único para aquelas entidades. Por exemplo, as tuplas na relação SUPPLIER representam fornecedores individuais, e os valores dos atributos S# identificam na realidade aqueles fornecedores, e não somente as tuplas que os representam. Esta interpretação leva-nos a impor a seguinte regra.

Regra de Integridade 1 (Integridade entidade)

Nenhum componente do valor de uma chave primária pode ser nulo.

O racional atrás desta regra é o seguinte. Por definição, todas as entidades têm que ser distinguíveis — isto é, têm que possuir uma identificação única de algum tipo. As chaves primárias preenchem a função de identificadores únicos em um banco de dados relacional. Um identificador (valor da chave primária) que fosse totalmente nulo seria uma contradição em termos; com efeito, ele estaria dizendo que existiria alguma entidade sem identificação única — isto é, não seria distinguível de outras entidades (e se duas entidades não forem distinguíveis uma da outra, elas por definição não são duas entidades, mas somente uma). Argumentação análoga dá a entender que identificadores *parcialmente* nulos devam também ser proibidos.

Considerações semelhantes nos levam à segunda regra de integridade. É comum que uma relação inclua referências a outra. Por exemplo, a relação AFTER (Fig. 4.5) inclui referências tanto à relação SUPPLIER como à relação PART, via seus atributos S# e P#. É claro que se uma tupla de AFTER contiver um valor de S#, *digamos s*, então deve existir em SUPPLIER uma tupla para o fornecedor *s* (senão, a tupla de AFTER estaria aparentemente se referindo a um fornecedor inexistente); o mesmo ocorre para peças. Podemos tornar essas noções precisas de seguinte maneira.

Primeiramente, vamos introduzir a noção de *domínio primário*. Um determinado domínio pode ser considerado como *primário* se e somente se existir alguma chave primária de atributo único naquele domínio. Por exemplo, nós podemos designar o domínio

PART_NUMBER como primário, estendendo sua definição (veja Fig. 4.3) como se segue:

```
DOMAIN PART_NUMBER CHARACTER (6) PRIMARY
```

Segundo, qualquer relação que inclua um atributo que esteja definido em um domínio primário (por exemplo, a relação AFTER), tem que obedecer à seguinte restrição.

Regra de Integridade 2 (Integridade Referencial)

Seja D um domínio primário, e seja R_1 uma relação com um atributo A que está definido em D . Então, em qualquer momento, cada valor de A em R_1 tem que ser (a) nulo, ou (b) igual a V , digamos, onde V é o valor da chave primária de alguma tupla em alguma relação R_2 (R_1 e R_2 não necessariamente distintos) com chave primária definida em D .

(Note-se que R_2 tem que existir, por definição de domínio primário. Note-se também que a restrição será trivialmente satisfeita se A for a chave primária de R_1 .)

Um atributo como A costuma ser chamado de *chave estrangeira*². Por exemplo, o atributo P# é uma chave estrangeira, pois seus valores são valores da chave primária da relação PART. As chaves, primária e estrangeira, fornecem meios para se representar relacionamentos entre tuplas; note-se, entretanto, que nem todos esses atributos de “relacionamentos” são chaves. Por exemplo, há um relacionamento (“localização”) entre peças e fornecedores, representado pelos atributos CITY das relações PART e SUPPLIER (veja Figs. 4.1 e 4.6), mas CITY não é uma chave estrangeira. (Elas poderia *tornar-se* uma chave estrangeira se fosse adicionada ao banco de dados uma relação tendo CITY como chave primária.)

Para concluir nossas discussões sobre chaves devemos mencionar que o acesso a uma relação a partir de uma DML relacional não deve estar restrito ao “acesso por chave primária”. Este item já foi ilustrado nos exemplos do Capítulo 3 (veja, por exemplo, a Fig. 3.2 e o Exemplo 3.5.2).

4.4 EXTENSÕES E INTENSÕES

Uma relação em um banco de dados relacional possui na verdade dois componentes, uma extensão e uma intensão, muito embora seja comum glosar-se este fato em discussões informais e usar-se o termo “relação” para significar ora um componente ora o outro.³

A *extensão* de uma dada relação é o conjunto de tuplas que aparecem naquela relação em um determinado instante. A extensão varia portanto com o tempo (isto é, muda quando tuplas são criadas, destruídas e atualizadas). Em outras palavras, extensão é o mesmo que nós estávamos anteriormente chamando de visão. As tabelas das Figs. 4.1 e 4.6 são exemplos de extensões (exceto pelo fato de aquelas tabelas também mostrarem a nomenclatura da estrutura intensional, discutida abaixo).

² Esta definição de chave estrangeira não é idêntica à dada em trabalhos escritos mais antigos (incluindo a segunda edição deste livro).

³ Se imaginarmos uma relação como sendo uma variável no sentido ordinário de programação, então a intensão é o tipo daquela variável e a extensão o seu valor.

A intensão de uma dada relação, em contraste, é independente do tempo. Ela é basicamente a parte *permanente* da relação; em outras palavras, corresponde ao que está especificado no esquema relacional. Portanto a intensão define todas as possíveis extensões. Mais precisamente, a intensão é a combinação de duas coisas: uma estrutura de nomes e um conjunto de restrições de integridade.

- A *estrutura de nomes* consiste do nome da relação mais os nomes de cada atributo (cada um com o nome do domínio associado).
- As *restrições de integridade* podem ser subdivididas em restrições chave, restrições referenciais e outras restrições.

Restrições chave

Restrições chave são as causadas pela existência de chaves candidatas. A intensão inclui uma especificação do(s) atributo(s) constituintes da chave primária, e também especificação do(s) atributo(s) constituintes das chaves alternativas, se existirem. Cada uma dessas especificações implica uma restrição de unicidade (pela definição de chave candidata); além disso, a especificação da chave primária implica uma restrição de não-nulos (pela Regra de Integridade 1).

Restrições referenciais

Restrições referenciais são as causadas pela existência de chaves estrangeiras. A intensão inclui (indiretamente) uma especificação de todas as chaves estrangeiras na relação. Cada uma dessas especificações implica uma restrição referencial (pela Regra de Integridade 2).

Outras restrições

Em teoria são possíveis muitas outras restrições. Um exemplo poderia ser “Se a cidade é London, então o valor do status tem que ser 20” (para a relação SUPPLIER).

4.5 RESUMO

Podemos definir um *banco de dados relacional* como sendo um banco de dados percebido pelo usuário como uma coleção de relações normalizadas de vários graus que se modificam ao longo do tempo. (Por “relações que se modificam ao longo do tempo” queremos significar as extensões das relações, que são variáveis ao longo do tempo.) Em outras palavras, o termo “banco de dados relacional” significa um banco de dados para o qual os operadores disponíveis ao usuário são os que operam sobre estruturas relacionais. Não significa necessariamente que os dados estejam armazenados sob a forma física de tabelas. A Fig. 4.7 mostra um exemplo de banco de dados relacional (extensão); ele consiste de três relações, S (a relação SUPPLIER da Fig. 4.6), P (PART, da Fig. 4.1), e SP (de AFTER, da Fig. 4.5). Este banco de dados é uma versão expandida dos dados de teste da Fig. 3.1. Nos próximos capítulos nós basearemos a maioria dos nossos exemplos neste banco de dados.

A Fig. 4.8 mostra um esquema para o banco de dados da Fig. 4.7. Os atributos receberam o mesmo nome do domínio básico (por exemplo, a especificação do atributo S# da relação S é um encurtamento da especificação “S#: DOMAIN S#”). Observe que o esquema inclui especificações explícitas de todas as chaves primárias e alternativas; essas especificações possibilitam ao DBMS manter as restrições chave. O esquema também especifica, indiretamente (como?), que os atributos S# e P# da relação SP são chaves estrangei-

ras, e portanto possibilita ao DBMS manter as restrições referenciais. Este esquema específico não inclui quaisquer outras restrições.

Como dado adicional podemos dizer que, em termos tradicionais, uma relação se parece com um *arquivo*, uma tupla com um *registro* (ocorrência, não tipo), e um atributo com um *campo* (tipo, não ocorrência). Entretanto, essas correspondências são no máximo aproximadas. Colocando de outra maneira, relações podem ser vistas como arquivos *altamente disciplinados* – disciplina no sentido de resultar em uma simplificação considerável.

S	S#	SNAME	STATUS	CITY	SP	S#	P#	QTY
	S1	Smith	20	London		S1	P1	300
	S2	Jones	10	Paris		S1	P2	200
	S3	Blake	30	Paris		S1	P3	400
	S4	Clark	20	London		S1	P4	200
	S5	Adams	30	Athens		S1	P5	100
						S1	P6	100
P	P#	PNAME	COLOR	WEIGHT	CITY	S2	P1	300
	P1	Nut	Red	12	London	S2	P2	400
	P2	Bolt	Green	17	Paris	S3	P2	200
	P3	Screw	Blue	17	Rome	S4	P2	200
	P4	Screw	Red	14	London	S4	P4	300
	P5	Cam	Blue	12	Paris	S4	P5	400
	P6	Cog	Red	19	London			

Fig. 4.7 O banco de dados fornecedores e peças: Visão relacional.

```

DOMAIN S#      CHARACTER (5) PRIMARY
DOMAIN SNAME   CHARACTER (20)
DOMAIN STATUS  NUMERIC (3)
DOMAIN CITY    CHARACTER (15)
DOMAIN P#      CHARACTER (6) PRIMARY
DOMAIN PNAME   CHARACTER (20)
DOMAIN COLOR   CHARACTER (6)
DOMAIN WEIGHT  NUMERIC (4)
DOMAIN QTY     NUMERIC (5)

RELATION S (S#,SNAME,STATUS,CITY)
        PRIMARY KEY (S#)
        ALTERNATE KEY(SNAME)
RELATION P (P#,PNAME,COLOR,WEIGHT,CITY)
        PRIMARY KEY (P#)
RELATION SP (S#,P#,QTY)
        PRIMARY KEY (S#,P#)

```

Fig. 4.8 O banco de dados fornecedores e peças: Esquema relacional.

rável nas estruturas dos dados com as quais o usuário tem que lidar, e consequentemente em uma simplificação correspondente nos operadores necessários para a manipulação (como foi demonstrado no Capítulo 3). Vamos concluir sumarizando informalmente os dispositivos mais importantes dos "arquivos" relacionais que os distinguem dos arquivos tradicionais, não disciplinados.

1. Cada "arquivo" contém sozinho um tipo de registro.
2. Cada ocorrência de registro em um dado "arquivo" têm o mesmo número de campos (em termos COBOL, OCCURS DEPENDING ON fica proscrito – isto é, não são permitidos grupos repetitivos).
3. Cada ocorrência de registro possui um identificador único.
4. Dentro de um "arquivo", as ocorrências de registros têm, ou uma ordenação não-específica, ou uma ordenação de acordo com os valores contidos nessas ocorrências. (O campo de ordenação [combinação] não é necessariamente a chave primária.) As relações definidas na Fig. 4.8 não têm ordenação específica.

EXERCÍCIOS

4.1 Este capítulo introduziu um grande número de termos novos. Em muitos casos o conceito básico já lhe é familiar, mas em geral os termos relacionais desses conceitos possuem uma definição mais precisa do que muitos dos termos mais tradicionais de processamento de dados. Nós organizamos esses novos termos abaixo em duas colunas; você deve ser capaz de dar as definições pelo menos para os que estão na coluna esquerda.

relação	grau
domínio	cardinalidade
atributo	relação unária
tupla	relação binária
relação normalizada	produto Cartesiano
chave primária	domínio único
banco de dados relacional	chave candidata
chave estrangeira	chave alternativa
extensão	ordenação controlada pelo valor
intensão	valor nulo

- 4.2 Defina um esquema para a sua solução do Exercício 3.1 (estrutura relacional para o banco de dados de pessoas e habilidades).
- 4.3 Sumarize as principais diferenças entre um arquivo relacional e um tradicional.
- 4.4 Estabeleça as Regras de Integridade 1 e 2.
- 4.5 Como você implementaria valores nulos? *Nota:* Muito poucos sistemas, relacionais ou outros, realmente suportam nulos. O que você acha disso?

REFERÊNCIAS E BIBLIOGRAFIA

4.1 E. F. Codd. "A Relational Model of Data for Large Shared Data Banks." *CACM* 13, nº 6 (junho de 1970).

O interesse atual na abordagem relacional é devido em grande parte ao trabalho de Codd, e este é o artigo que disparou a maioria das atividades subsequentes no campo. Ele contém uma explanação sobre a estrutura relacional, definições de algumas operações da álgebra relacional e uma discussão sobre redundância e consistência. É um artigo original. A explicação sobre estrutura relacional dada no presente capítulo foi, naturalmente, baseada em [4.1], mas inclui um grande número de refinamentos incorporados desde 1970.

- 4.2 E. F. Codd. "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks." IBM Research Report RJ 599 (agosto de 1969).
Uma versão anterior de [4.1].
- 4.3 E. F. Codd. "Understanding Relations." Série contínua de artigos na FDT (boletim trimestral da ACM Special Interest Group on Management of Data [SIGMOD, anteriormente SIGFIDET]) começando no volume 5, nº 1 (junho de 1973).
- 4.4 E. F. Codd (ed.). "Relational Data Base Management: A Bibliography." IBM Research Laboratory, San Jose, CA 95193 (agosto de 1975).
Uma relação abrangente (mas sem anotações) de referências atinentes à abordagem relacional até 1975. As referências estão organizadas sob os seguintes cabeçalhos.
- Modelos e teoria
 - Linguagens e fatores humanos
 - Implementações
 - Tecnologia de implementação
 - Autorização, visões e concorrência
 - Controle de integridade
 - Aplicações
 - Interferência dedutiva e raciocínio aproximado
 - Suporte à linguagem natural
 - Conjuntos e relações anteriores a 1969.
- 4.5 D. D. Chamberlin. "Relational Data Base Management: A Survey." *ACM Comp. Surv.* 8, nº 1 (março de 1976).
Um estudo. Este artigo está organizado sobre a bibliografia [4.4], incluída como um apêndice.
- 4.6 E. F. Codd. "Extending the Database Relational Model to Capture More Meaning." *ACM Transactions on Database Systems* 4, nº 4 (dezembro de 1979).
- 4.7 W. Kim. "Relational Database Systems." *ACM Comp. Surv.* 11, nº 3 (setembro de 1979).
Uma descrição sobre cerca de 30 sistemas relacionais, na sua maioria de natureza experimental, sob os seguintes títulos.
- Levantamento
 - Estruturas de armazenamento e caminhos de acesso
 - Otimização do interface relacional
 - Visões e instantâneos do usuário
 - Controle seletivo de acesso
 - Controle de integridade
 - Controle de concorrência
 - Recuperação
 - Monitoração do desempenho e da resposta ao usuário

5

A Arquitetura do Sistema R

5.1 HISTÓRICO

Ocorreram muitas implementações da abordagem relacional desde a publicação dos artigos originais de Codd [4.1, 4.2] em 1969 e 1970. Até 1974, no entanto, os únicos sistemas implementados eram algo experimentais, não estando disponíveis comercialmente (embora uns poucos sistemas comerciais oferecessem alguns dispositivos relacionais). Além disso, a maior parte dessas primeiras implementações tendia a se concentrar em aspectos específicos do sistema, tais como suporte a consultas não planejadas, excluindo outros aspectos; por exemplo, poucos desses sistemas, se é que algum, suportavam recuperação do banco de dados.

Hoje a situação está mudando em pelo menos dois pontos. Primeiro, os fornecedores de *software* começaram a incluir sistemas relacionais entre seus produtos; como exemplos temos NOMAD, MAGNUM, e Query By Example. Segundo, a atividade em torno de protótipos continuou, mas esses protótipos tornaram-se consideravelmente mais ambiciosos, tentando abranger parcela muito maior do problema total; como exemplos temos INGRES e o System R. Nós escolhemos o sistema R como nosso exemplo fundamental de sistema relacional, pois ele incorpora a maioria dos princípios que desejamos discutir. Por isso os próximos capítulos oferecem uma descrição bastante abrangente do Sistema R; entretanto, o leitor deve ter em mente que o objetivo desses capítulos é a discussão das idéias relacionais em geral, além de cobrir especificamente o Sistema R. Serão descritos dispositivos específicos de outros sistemas quando apropriado — por exemplo, para ilustrar um conceito relacional não suportado pelo Sistema R.

O Sistema R foi projetado e desenvolvido no período de 1974 a 1979 no IBM San Jose Research Laboratory [5.1, 5.2, 5.3]. É um protótipo, não um produto. Entretanto, o objetivo global durante o desenvolvimento foi o de tornar o sistema “operacionalmente completo”; isto é, a finalidade do protótipo foi a de “demonstrar que é possível construir um sistema relacional que pode ser usado no ambiente real para resolver problemas reais, com um desempenho pelo menos comparável ao dos sistemas existentes” [5.1]. Assim sendo, o Sistema R oferece não somente os elementos básicos de um banco de dados re-

lacional (como nos protótipos anteriores), mas também muitos dispositivos adicionais, incluindo:

- gerenciamento da recuperação do banco de dados
- controle automático de concorrência
- um mecanismo flexível de autorização
- independência de dados
- definição dinâmica do banco de dados
- dispositivos para ajuste e melhoria da forma de uso

(Veja as referências, especialmente [5.1] e [5.3].) Embora possa parecer absurdo dizer-se que somente um sistema relacional poderia possuir todas essas características desejáveis, é lícito declarar-se que um sistema erigido sobre fundações relacionais possui vantagens significativas sobre seus competidores: não considerando outras razões, o formalismo relacional oferece uma estruturação conceitual que permite uma formulação mais abrangente dos problemas, tornando-os geralmente mais fáceis de administrar.

O Sistema R corre no sistema IBM/370, usando VM/CMS ou MVS como sistema operacional básico. Suporta um banco de dados relacional, isto é, um banco de dados no qual os dados são percebidos pelo usuário na forma de tabelas. (O Sistema R recorre a tabelas ao invés de relações; registros ou linhas ao invés de tuplas; e campos ou colunas ao invés de atributos. Nós faremos o mesmo no contexto do Sistema R.) Todo acesso a este banco de dados é feito via uma sublinguagem chamada SQL (anteriormente escrita SEQUEL, e normalmente pronunciada como se ainda fosse SEQUEL). A versão original do SQL [5.4] foi baseada em uma linguagem anterior chamada SQUARE [5.6]. As duas linguagens são essencialmente a mesma, mas a SQUARE usa uma sintaxe bem mais matemática, enquanto a SQL é muito mais parecida com o inglês. Foi também montado um protótipo de implementação da versão original do SQL no IBM San Jose Research Laboratory [5.7, 5.8]. Além disso, foram feitas experiências quanto à possibilidade de uso da linguagem, usando estudantes universitários como objetos [5.9]. Como resultado desse trabalho, foram feitas várias melhorias à linguagem.

SQL é a abreviatura de “structured query language” (linguagem estruturada de consulta) [5.5]. Mas SQL é mais do que somente uma linguagem de consulta, sem que isto se oponha ao “query” no seu nome. Primeiro, ela oferece não somente funções de recuperação, mas também toda a gama de operações de atualização, além de outros elementos que discutiremos em capítulos subsequentes. Segundo, ela pode ser usada tanto a partir de um terminal como, na forma de “SQL embutida”, a partir de um programa de aplicação, *batch* ou *on-line*, escrito em COBOL ou PL/I. O nível geral da linguagem é comparável ao da álgebra relacional; em outras palavras, seus operadores geralmente funcionam em termos de conjuntos ao invés de registros individuais, não contendo qualquer referência explícita a caminhos de acesso.

5.2 ARQUITETURA

É possível estabelecer-se uma correspondência entre as funções oferecidas pelo Sistema R e a arquitetura ANSI/SPARC. A correspondência não é totalmente direta, como veremos, mas ainda assim é útil como auxílio para o entendimento. A Fig. 5.1 procura mostrar essa correspondência; ela representa o Sistema R, *como percebido por um usuário individual*, sob uma perspectiva ANSI/SPARC.

Explicaremos a Fig. 5.1 a seguir.

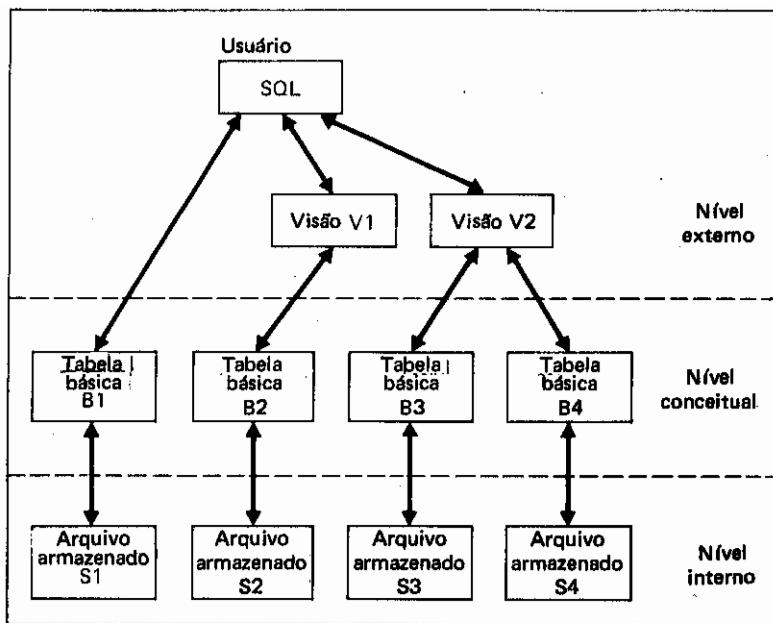


Fig. 5.1 O Sistema R visto por um usuário individual.

1. O equivalente mais próximo do “tipo de registro conceitual” ANSI/SPARC é a *tabela básica*. Uma tabela básica é uma tabela que tem existência independente – isto é, ela não é uma “visão” derivada de outras tabelas básicas (veja abaixo). Cada tabela básica está representada na memória por um arquivo armazenado distinto.

2. Uma tabela vista pelo usuário pode ser uma tabela básica ou uma *visão*. O termo “visão” é usado no Sistema R com um significado muito específico: uma visão é uma tabela que não existe por si mesma, mas é derivada de uma ou mais tabelas básicas. Por exemplo, se o banco de dados inclui uma tabela básica S#, com campos S#, SNAME, STATUS, e CITY, então nós podemos definir uma visão chamada LONDON_SUPPLIER, digamos, com campos S#, SNAME, STATUS, derivada de S, selecionando os campos que têm CITY = ‘LONDON’ e depois projetando o campo CITY. Observe a diferença entre a visão do Sistema R e uma visão externa como definida no Capítulo 1 – no Sistema R, o usuário estará tipicamente interagindo com diversas visões (e tabelas básicas) ao mesmo tempo, enquanto que no Capítulo 1 nós definimos uma visão externa como a *totalidade* dos dados vistos por algum usuário. Nós usaremos “visão” no sentido do Sistema R, ao invés do sentido da ANSI/SPARC, sempre que estivermos lidando especificamente com o Sistema R. Em outros casos deixaremos que o contexto defina o nosso significado.

3. No nível interno, como dito anteriormente, cada tabela básica está representada na memória por um arquivo armazenado distinto, isto é, por um conjunto com nome de ocorrências de registros armazenados, todos do mesmo tipo. (“Arquivo armazenado” não é um termo do Sistema R.) Uma linha na tabela básica corresponde a uma ocorrência de

registro armazenado no arquivo armazenado. Um dado arquivo armazenado pode ter qualquer quantidade de *índices* associados a ele. Usuários que se encontrem acima do nível interno poderão ter conhecimento desses índices, mas não poderão referir-se diretamente a eles nas solicitações de acesso aos dados. Os índices podem ser criados ou destruídos a qualquer momento, sem afetar os usuários (a não ser em desempenho).

4. SQL é a sublinguagem de dados do Sistema R. Como tal, inclui tanto uma linguagem de definição de dados (DDL) como uma linguagem de manipulação de dados (DML). Conforme já mencionado, a DML pode operar tanto no nível externo quanto no conceitual. Semelhantemente, a DDL pode ser usada para definir objetos no nível externo (visões) no nível conceitual (tabelas básicas) e até no nível interno (índices). Além disso, a SQL também oferece elementos para “controle de dados”, isto é, esses elementos não se enquadram realmente nas classificações da DDL ou da DML como pertencentes a estas. Um exemplo de um desses elementos é a instrução para GRANT (conceder) certos direitos de acesso a outro usuário.

5. Usuários de programação de aplicações podem ter acesso ao banco de dados por meio de instruções de COBOL ou PL/I via a “SQL embutida”. A SQL embutida representa uma “união frouxa” entre a SQL e a linguagem principal (veja o Capítulo 1). Praticamente qualquer instrução da linguagem SQL pode ser embutida. Além disso, há certas instruções especiais (a serem discutidas no Capítulo 8) fornecidas somente para o uso embutido.

6. Uma aplicação *on-line* especial, fornecida com o sistema, é o User-Friendly Interface, ou UFI. O UFI permite que o usuário *on-line* tenha acesso ao banco de dados usando a SQL como uma linguagem interativa de consulta, isto é, o UFI aceita instruções do terminal, passa-as para o Sistema R para execução, e depois retorna o resultado ao terminal se apropriado. O UFI também fornece vários comandos especiais para controlar a tela de saída, modificar e reexecutar instruções SQL previamente entradas, e assim por diante.

Nas nossas discussões sobre a linguagem SQL vamos supor um ambiente UFI — isto é, vamos supor que as instruções estão sendo colocadas a partir de um terminal *on-line*. O ambiente de “SQL embutida” será discutido em detalhes no Capítulo 8.

Componentes do Sistema R

O Sistema R consiste de dois subsistemas principais: o sistema de pesquisa à memória (Research Storage System, ou RSS), e o sistema relacional de dados (Relational Data System, ou RDS). Basicamente, o RDS fornece o interface do usuário externo, suportando estruturas tabulares de dados e operadores sobre essas estruturas (a linguagem SQL), e o RSS fornece um interface do registro armazenado para o RDS (compare a estrutura deste sistema com a estrutura ilustrada na Fig. 2.1). Abaixo estaremos considerando cada um destes dois componentes com alguns detalhes adicionais.

Sistema de Pesquisa à Memória (RSS)

O RSS é essencialmente um método de acesso poderoso. Sua função primária é a de manusear todos os detalhes do nível físico e apresentar ao seu usuário um interface chamado Interface de Pesquisa à Memória (Research Storage Interface, ou RSI), que é o interface do registro armazenado há pouco mencionado. Normalmente o usuário do RSS não é o usuário direto, mas sim codificação gerada pelo RDS ao compilar alguma instrução SQL

(veja abaixo). O RSI foi especificamente projetado para se comportar como um bom alvo para o compilador SQL.

O dado objeto básico suportado pelo RSI é o arquivo armazenado, isto é, a representação interna de uma tabela básica. As linhas da tabela são representadas pelos registros do arquivo; entretanto, os registros armazenados dentro de um arquivo armazenado não têm que estar fisicamente adjacentes na memória. O RSS também suporta uma quantidade arbitrária de índices relativos a qualquer dado arquivo armazenado. O RSI fornece operadores para a pesquisa através de um arquivo armazenado na sequência "sistema" (isto é, definida pelo RSS), e em uma sequência de acordo com qualquer índice especificado. (Por sequência de sistema queremos significar uma sequência que não foi especificada externamente, mas sim determinada pela maneira como o RSS representa o arquivo armazenado no meio físico.) O usuário do RSI precisa conhecer que arquivos armazenados e índices existem, e tem que especificar o caminho de acesso (sequência de algum índice ou do sistema) a ser usado em qualquer solicitação de acesso RSI.

O RSS também oferece funções adicionais, que serão discutidas no Capítulo 10.

Sistema Relacional de Dados (RDS)

O RDS, por seu turno, consiste de dois componentes: um pré-compilador e um sistema de controle em tempo de execução.

O *pré-compilador* é um compilador para a linguagem SQL. Suponhamos que o programador de aplicações escreveu um programa P que inclui algumas instruções embutidas da SQL. Para fixar nossa idéia, suponhamos que P está escrito em PL/I (o quadro é essencialmente o mesmo para COBOL). Antes que P possa ser compilado pelo compilador PL/I da forma convencional, tem que ser primeiramente processado pelo pré-compilador RDS. O processo de pré-compilação está ilustrado na Fig. 5.2. A instrução "\$ SELECT. . ." naquela figura é um exemplo de uma instrução SQL embutida no PL/I.

A pré-compilação ocorre da seguinte maneira.

1. O pré-compilador esquadra o programa fonte P e localiza as instruções SQL embutidas (as que têm um prefixo \$).
2. Para cada instrução encontrada, o pré-compilador decide qual a estratégia para implementar aquela instrução em termos de operações RSI. Este processo é conhecido como *otimização*. A otimização baseia-se no conhecimento que o pré-compilador tem dos caminhos de acesso disponíveis no RSI. (Lembre-se de que as instruções SQL tais como SELECT não incluem qualquer referência a esses caminhos de acesso.) Tomadas as decisões, o pré-compilador gera uma rotina em linguagem de máquina do Sistema/370 (incluindo chamadas ao RSS) que implementará a estratégia escolhida. O conjunto de todas essas rotinas reunidas constitui o *módulo de acesso* desse programa P. O módulo de acesso em si fica armazenado no banco de dados.

3. O pré-compilador substitui cada uma das instruções SQL embutidas originais por uma instrução ordinária do PL/I para CALL XRDI (XRDI é o nome do componente de controle em tempo de execução do RDS).

O programa fonte P modificado pode agora ser compilado pelo compilador PL/I da forma normal.

A explicação anterior aplica-se ao caso da SQL *embutida*. O processo é essencialmente semelhante para o caso da linguagem *on-line* (submetida via o User-Friendly Interface, UFI), com exceção de que a execução ocorre logo que a pré-compilação se completa. Em outras

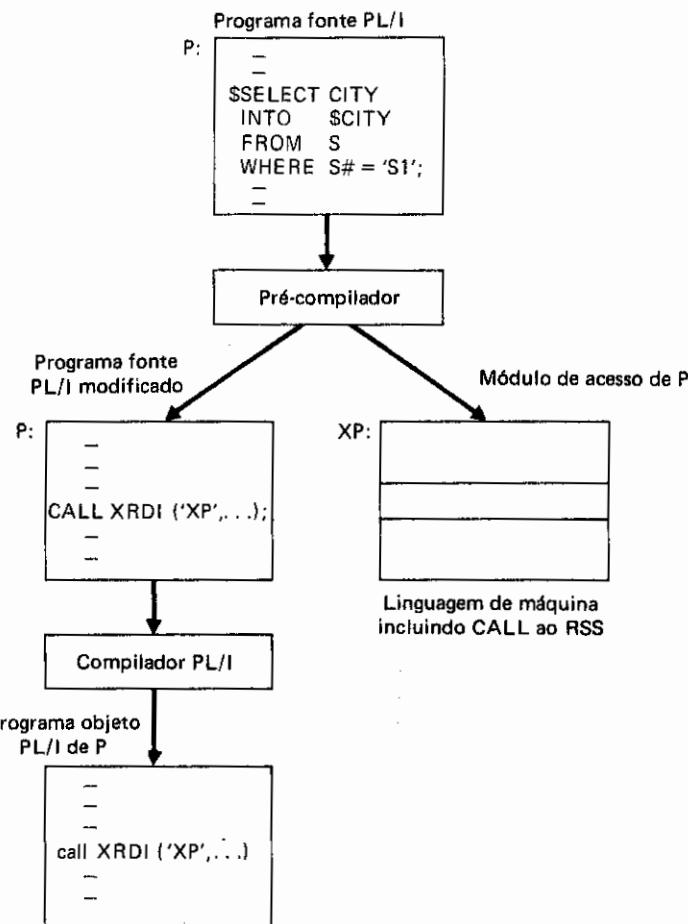


Fig. 5.2 Pré-compilação de um programa PL/I com SQL embutida.

palavras, o UFI lê as instruções SQL do terminal, passa-as ao pré-compilador, e então executa imediatamente o módulo de acesso compilado (via XRDI, naturalmente). O compilador PL/I não é envolvido, pois (a) a instrução original permanece isolada (isto é, não tem nenhuma instrução PL/I adjacente) e também (b) o pré-compilador gera diretamente o código de máquina (isto é, não trabalha em “cascata” através do PL/I).

O sistema de controle em tempo de execução (XRDI) fornece o ambiente para a execução de um programa de aplicação que tenha passado pelo processo acima descrito. Consideremos novamente a Fig. 5.2. Quando o programa objeto P é executado, ele eventualmente atingirá a instrução “call XRDI” que substitui a instrução SQL. O controle será passado para o sistema de controle em tempo de execução do RDS, que carregará o módulo de acesso de P (se necessário), e depois chamará a seção apropriada daquele módulo de acesso, ou seja, a seção que foi gerada para corresponder à instrução original SQL.

Programa objeto PL/I de P

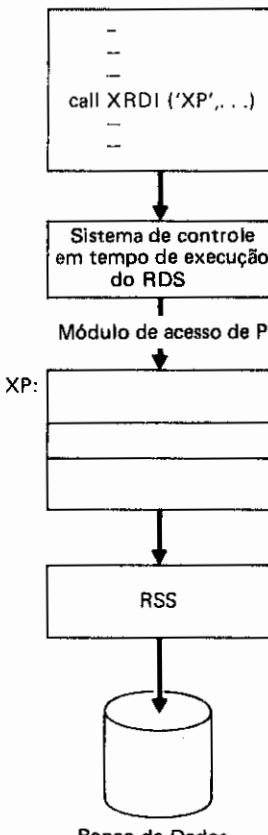


Fig. 5.3 Execução de um programa pré-compilado.

SELECT. Esta seção, por seu lado, invocará diversas operações RSS para executar as ações requeridas pela instrução SELECT original. Veja a Fig. 5.3.

O sistema de controle em tempo de execução executa uma série de outras funções que serão abordadas em capítulos posteriores.

REFERÊNCIAS E BIBLIOGRAFIA

- 5.1 M. M. Astrahan et al. "System R: Relational Approach to Database Management". *ACM Transactions on Database Systems* 1, nº 2 (junho de 1976).
- 5.2 M. M. Astrahan et al. "System R, A Relational Database Management System." *IEEE Computer Society: Computer* 12, nº 5 (maio de 1979).
- 5.3 M. W. Blasgen et al. "System R: An Architectural Update." IBM Research Report RJ2581 (julho de 1979).
- 5.4 D. D. Chamberlin and R. F. Boyce. "SEQUEL: A Structured English Query Language." *Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control*.

- 5.5 D. D. Chamberlin et al. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control." *IBM J. R & D* 20, nº 6 (novembro de 1976).
- 5.6 R. F. Boyce, D. D. Chamberlin, W. F. King III, and M. M. Hammer. "Specifying Queries as Relational Expressions: SQUARE." Versões deste artigo apareceram em *Proc. ACM SIGPLAN/SIGIR Interface Meeting on Programming Languages and Information Retrieval* (novembro de 1973); *Proc. IFIP TC-2 Working Conference on Data Base Management Systems* (eds., Klimbie and Koffeman) (North-Holland, 1974); and *CACM 18*, nº 11 (novembro de 1975).
- 5.7 M. M. Astrahan and R. A. Lorie. "SEQUEL-XRM, a Relational System." *Proc. ACM Pacific Conference, San Francisco* (abril de 1975). Disponível na ACM Golden Gate Chapter, P. O. Box 24055, Oakland, California 94623.
- 5.8 M. M. Astrahan and D. D. Chamberlin. "Implementation of a Structured English Query Language." *CACM 18*, nº 10 (outubro de 1975).
- 5.9 P. Reisner, R. F. Boyce, and D. D. Chamberlin. "Human Factors Evaluation of Two Data Base Query Languages – SQUARE and SEQUEL." *Proc. NCC 44* (maio de 1975).
- 5.10 D. D. Chamberlin. (ed., S. M. Deen and P. Hammersley). "A Summary of User Experience with the SQL Data Sublanguage." *Proc. International Conference on Data Bases, Aberdeen, Scotland* (julho de 1980). Também disponível como um IBM Research Report RJ2767 (abril de 1980).
- 5.11 M. M. Astrahan et al. "A History and Evaluation of System R." IBM Research Report RJ2843 (junho de 1980).

6

Estrutura de Dados do Sistema R

6.1 INTRODUÇÃO

A estrutura primária de dados no Sistema R é a tabela básica. Neste capítulo consideraremos as tabelas básicas em detalhes; mostraremos como as tabelas básicas são criadas e destruídas usando-se os elementos da DDL da SQL, e faremos comparações e contrastes entre a estrutura tabular e a estrutura relacional definida no Capítulo 4. Como referência, encontramos na Fig. 6.1 um equivalente em Sistema R do esquema de fornecedores e peças da Fig. 4.8. (O termo esquema não é utilizado no Sistema R.)

```
CREATE TABLE S ( S#          (CHAR(5), NONULL),
                  SNAME      (CHAR(20)),
                  STATUS     (SMALLINT),
                  CITY       (CHAR(15)) )

CREATE TABLE P ( P#          (CHAR(6), NONULL),
                  PNAME      (CHAR(20)),
                  COLOR      (CHAR(6)),
                  WEIGHT    (SMALLINT),
                  CITY       (CHAR(15)) )

CREATE TABLE SP ( S#          (CHAR(5), NONULL),
                  P#          (CHAR(6), NONULL),
                  QTY        (INTEGER) )
```

Fig. 6.1 Definição SQL das tabelas S, P e SP.

6.2 TABELAS BÁSICAS

Como explicado no Capítulo 5, uma tabela básica é aquela que tem existência independente. Ela está representada no banco de dados físico por um arquivo armazenado. Uma

tabela básica pode ser criada a qualquer momento através da execução da instrução CREATE TABLE da SQL DDL, que tem como forma geral

```
CREATE TABLE base-table-name  
  ( field-definition [, field-definition ] ... )  
  [ IN SEGMENT segment-name ]
```

onde uma definição de campo, por sua vez, tem a forma

```
  field-name ( data-type [,NONULL] )
```

(Excetuando-se definição explícita do contrário, são usados colchetes quadrados ao longo deste livro dentro de definições de sintaxe para indicar que o contido entre esses colchetes pode ser omitido.)

A execução bem-sucedida da instrução CREATE TABLE faz com que seja criada uma nova tabela vazia no segmento especificado com o nome especificado da tabela básica e com as definições de campos especificadas. (Explicaremos segmentos e definições de campos adiante.) O usuário pode agora começar a entrar com seus dados na tabela usando a instrução SQL INSERT (parte da DML da SQL). Alternativamente, o usuário pode recorrer ao System R loader, um utilitário fornecido com o sistema, para executar esta função; detalhes sobre o utilitário loader estão fora do escopo deste livro.

Segmentos Um banco de dados do Sistema R é particionado em um conjunto de *segmentos* separados. (Este particionamento encontra-se discutido em maior profundidade no Capítulo 10.) Os segmentos fornecem um mecanismo para o controle da alocação de memória e do uso compartilhado de dados entre os usuários. Qualquer dada tabela básica encontra-se totalmente contida dentro de um único segmento; quaisquer índices referentes àquela tabela básica também estarão contidos no mesmo segmento. Entretanto, um determinado segmento pode conter diversas tabelas básicas (e seus índices).

Um segmento *público* contém dados compartilhados que podem receber acesso simultâneo de múltiplos usuários. Um segmento *privado* contém dados que só podem ser utilizados por um usuário de cada vez (ou dados que nunca são compartilhados). A instrução CREATE TABLE opcionalmente especifica o segmento que deverá conter a nova tabela básica; se o segmento não for especificado, a tabela será criada em um segmento *privado* pertencente ao usuário que emitiu o CREATE TABLE. Por isso, na Fig. 6.1, cada CREATE TABLE teria que incluir a especificação "IN SEGMENT SHARED_SEG" (digamos), onde SHARED_SEG é um segmento público, caso o banco de dados de fornecedores e peças desse estiver concorrentemente disponível a múltiplos usuários.

Campos Cada *definição de campo* na CREATE TABLE inclui três itens: um nome de campo, um tipo de dado do campo e (opcionalmente) uma especificação NONULL. Naturalmente, o nome do campo tem que ser único dentro do banco de dados. Os tipos de dados permissíveis são os seguintes:

CHAR (<i>n</i>):	seqüência de caracteres de comprimento fixo,
CHAR (<i>n</i>) VAR:	seqüência de caracteres de comprimento variável,
INTEGER:	uma palavra de inteiro binário,
SMALLINT:	meia palavra de inteiro binário,
FLOAT:	palavra dupla de número em ponto flutuante.

O Sistema R suporta o conceito de valor *nulo* de campo. De fato, qualquer campo pode conter um valor nulo *exceto* se a definição daquele campo na CREATE TABLE incluir

especificamente a especificação NONULL. Nulo é um valor especial usado para representar “valor desconhecido” ou “valor que não se aplica”. Por exemplo, um registro de embarque poderia conter um valor de QTY nulo (sabemos que o embarque existe, mas desconhecemos a quantidade remetida); ou um registro de fornecedor poderia conter um valor nulo de STATUS (talvez “STATUS” não se aplique aos fornecedores de Paris por alguma razão). Não discutiremos em detalhes as propriedades do nulo aqui, contentando-nos em saber que (a) expressões aritméticas nas quais um dos operandos é um nulo são avaliadas como nulas, e (b) comparações nas quais um dos comparandos é nulo são avaliadas como “desconhecidas” [veja o exemplo 7.2.22 para uma ilustração de (b)].

Da mesma forma como uma nova tabela pode ser criada a qualquer momento, uma tabela básica existente pode ser *expandida* a qualquer momento pela adição de uma nova coluna (campo) à direita¹:

```
EXPAND TABLE base-table-name  
        ADD FIELD field-name ( data-type )
```

Por exemplo,

```
EXPAND TABLE SP  
        ADD FIELD DATE ( CHAR ( 6 ) )
```

Esta instrução adiciona o campo DATE à tabela SP. Todos os registros existentes em SP são expandidos de três para quatro campos: o valor do novo campo é nulo em todos os casos (não é permitida a especificação NONULL em EXPAND TABLE). Note, incidentalmente, que a expansão dos registros existentes que acabamos de descrever não significa que os registros do banco de dados sejam realmente atualizados nesse momento; somente muda a sua descrição armazenada. (Esta descrição está armazenada no dicionário do Sistema R – veja Capítulo 1. Nós discutiremos o dicionário do Sistema R com mais alguns detalhes no Capítulo 7.)

Também é possível *destruir* uma tabela básica existente a qualquer momento:

```
DROP TABLE base-table-name
```

Todos os registros da tabela básica especificada são removidos, todos os índices e visões relativos àquela tabela são destruídos (isto é, suas descrições são removidas do dicionário, e o espaço de armazenagem respectivo colocado disponível).

6.3 ÍNDICES

Da mesma forma que as tabelas básicas, os índices são criados ou desfeitos usando-se a DDL SQL. No entanto, as *únicas* instruções da SQL que tratam de índices são CREATE INDEX e DROP INDEX, além de algumas instruções de controle de dados; as outras instruções – em particular instruções de acesso a dados tais como SELECT – deliberadamente não incluem essas referências. A decisão de usar ou não um índice em resposta a uma solicitação específica de dado não é tomada pelo usuário, mas sim pelo Sistema R (na verdade, pelo componente de otimização do pré-compilador RDS).

CREATE INDEX tem como forma geral

```
CREATE [ UNIQUE ] INDEX index-name ON base-table-name  
        ( field-name [ order ] [ , field-name [ order ] ]... )
```

¹ No Sistema R, a ordem da esquerda para a direita das colunas em uma tabela básica é significativa (contrasta com o último parágrafo da Seção 4.1).

onde “order” (ordem) pode ser ASC (ascendente) ou DESC (descendente). Se não for especificado ASC nem DESC, será assumido ASC à revelia. A seqüência de nomes de campos da esquerda para a direita na instrução CREATE INDEX corresponde à ordenação de maior para menor da forma usual. Por exemplo, a instrução criará um índice de nome X

```
CREATE INDEX X ON T (A, B, C)
```

na tabela básica T, cujas entradas estão ordenadas pelos valores crescentes de C dentro dos valores crescentes de B dentro dos valores crescentes de A.²

Uma vez criados, os índices são mantidos automaticamente (pelo RSS) para que reflitam atualizações sobre a tabela básica indexada, até que o índice seja destruído.

A opção UNIQUE na instrução CREATE INDEX especifica que não será permitido que dois registros na tabela básica indexada assumam o mesmo valor no campo ou combinação de campos indexado (ao mesmo tempo). Esta é a única forma de se especificar que não são permitidos valores duplicados em algum campo ou combinação de campos. Por isso, se quisermos garantir a unicidade das chaves primárias no banco de dados de fornecedores e peças, temos que acrescentar instruções como as que se seguem à Fig. 6.1:

```
CREATE UNIQUE INDEX XS ON S (S#)
CREATE UNIQUE INDEX XP ON P (P#)
CREATE UNIQUE INDEX XSP ON SP (S#, P#)
```

(Tal como nas tabelas básicas, os índices podem ser criados e destruídos a qualquer momento. No exemplo, temos que garantir a criação dos índices XS, XP, e XSP antes que qualquer dado seja colocado nas tabelas básicas S, P, e SP; caso contrário, as restrições de unicidade já poderão ter sido violadas. Uma tentativa de criar um índice único em tabela que correntemente não satisfaça à restrição de unicidade irá falhar.)

A instrução para destruir um índice é:

```
DROP INDEX index-name
```

O índice será destruído; isto é, sua descrição removida do dicionário e o espaço de armazenamento colocado disponível. Caso algum programa pré-compilado possua um módulo de acesso (veja o Capítulo 5) que dependa do índice destruído, tal módulo de acesso será marcado como “inválido” pelo sistema de controle em tempo de execução do RDS. Na próxima vez que aquele módulo for chamado, o Sistema R irá automaticamente recompilar o programa original, gerando um módulo de acesso substituto que suporte as instruções SQL originais sem usar o índice agora inexistente. Este processo é totalmente transparente ao usuário.

6.4 DISCUSSÃO

O fato de poderem as instruções SQL DDL ser executadas a qualquer momento torna o Sistema R muito flexível. Na maioria dos sistemas convencionais, a adição de novos tipos de objetos ao banco de dados não é uma operação que possa ser feita suavemente; tipicamente envolve uma paralisação de todo o sistema, execução de algum utilitário especial para compilar a definição modificada do banco de dados (esquema), descarga e recarga de todo o banco de dados, ou pelo menos de alguma porção deste. Nesses sistemas torna-

²

No que se refere aos índices, valores nulos são considerados como sendo (a) todos iguais entre si, e (b) maiores do que qualquer valor não-nulo.

se altamente desejável executar o processo de definição dos dados de uma vez por todas antes do início da carga e uso do banco de dados, o que significa que (a) pode exigir mais tempo para que se tenha o sistema instalado e operacional, e (b) pode ser difícil e caro corrigir erros de projeto.

No Sistema R, em contraste, é possível criar e carregar umas poucas tabelas básicas e começar a usá-las imediatamente. É também possível experimentar os efeitos de ter ou não certos índices, e adicionar novas tabelas e novos campos aos poucos, tudo sem afetar as aplicações existentes em nada (exceto quanto ao desempenho, naturalmente). Além disso, como veremos no Capítulo 9, é também possível sob certas circunstâncias substituir uma tabela básica existente por duas ou mais (menores), novamente sem afetar os usuários existentes, fornecendo a esses usuários uma *visão* que é idêntica à da tabela básica original, mas definida em termos das novas tabelas menores. De forma resumida, não é necessário executar todo o processo de projeto do banco de dados antes de retirar dele algum trabalho útil, nem é necessário acertar em tudo da primeira vez.

Vamos concluir este capítulo comparando a estrutura de dados do Sistema R com a estrutura relacional definida no Capítulo 4. Deve ficar claro que há algumas diferenças entre as duas. Vamos summarizar os itens diferentes aqui, e depois discuti-los adiante. No Sistema R,

- não são suportados domínios
- obrigatoriedade de unicidade de chave candidata é opcional
- obrigatoriedade da Regra de Integridade 1 (integridade entidade) é opcional
- a Regra de Integridade 2 (integridade referencial) não é obrigatória

O objetivo desta lista não é o de criticar o Sistema R. (Na verdade, esses dispositivos estão omitidos na maioria dos sistemas relacionais.) As definições do Capítulo 4 incorporaram diversos refinamentos (especialmente as regras de integridade) que não haviam sido amplamente divulgados no período em que o Sistema R estava sendo projetado, de forma que as omissões não chegam a causar surpresa. A questão é — quão importantes são? Qualquer tentativa para dar resposta a esta questão será certamente marcada como subjetiva; apesar disso, podemos dizer que os três primeiros itens da lista não são muito importantes, mas o último item (falta de suporte para a Regra de Integridade 2) é significativo.³

Não são suportados domínios

O relacionamento entre um determinado fornecedor (digamos) e os embarques daquele fornecedor é estabelecido pelo aparecimento do mesmo valor no campo S# do registro S envolvido, e no campo S# dos registros SP correspondentes. Entretanto, este relacionamento não está explicitamente definido no Sistema R; ele está na mesma categoria do “relacionamento” entre, digamos, uma peça pesando 25 quilos e um embarque envolvendo a remessa de 25 itens, no que tange ao Sistema R. Uma consequência prática é que o sistema não tem como detetar o fato de que uma solicitação em que, digamos, a quantidade de peças embarcadas seja igual ao peso de uma peça é sem sentido.

³ Por outro lado, pode-se contestar o suporte à Regra de Integridade 2 na base de que ela leva a um aumento da complexidade tanto para o sistema como para o usuário. Por exemplo, o sistema tem que estar preparado para verificar e rejeitar certos tipos de atualização, e o usuário tem que estar preparado para manusear algumas situações adicionais de erro.

Os domínios são também relevantes para a Regra de Integridade 2; veja o final desta seção.

Obrigatoriedade de unicidade da chave candidata é opcional

O Sistema R não conhece as chaves (candidata, primária, alternativa, ou estrangeira). Na falta de especificações em contrário, os campos e combinações de campos *não* possuem a propriedade de unicidade; em geral, portanto, uma tabela básica não é o mesmo que uma relação, pois ela pode conter linhas duplicadas. Mas o Sistema R oferece um mecanismo, o índice único, pelo qual pode ser forçada a unicidade se desejado. Note, entretanto, que tal obrigatoriedade está amarrada à provisão de um caminho de acesso. Do ponto de vista de arquitetura, teria sido mais adequado separar as duas especificações. É verdade que, se um índice for provido por uma chave candidata, então esse índice tem que ser “único”; mas, no fim de tudo, o fato de um campo ser uma chave candidata não significa necessariamente que um caminho de acesso ordenado seja requerido naquele campo, nem é um índice único a única maneira de se forçar a unicidade. Além disso, no caso da tabela básica SP (Fig. 6.1), a unicidade de chave poderia ser forçada ou por um índice em (S#, P#), nessa ordem, ou por um índice em (P#, S#), na ordem reversa; com base em que escolheríamos uma ou outra ordem?

Obrigatoriedade da Regra de Integridade 1 (identidade entidade) é opcional

A Regra de Integridade 1 proíbe valores nulos nos campos da chave primária (para detalhes, veja o Capítulo 4). O Sistema R forçará esta restrição desde que seja especificado NONULL para os campos em questão. (Como um aparte, observemos que se forem permitidos nulos em um campo no qual está definido um índice único, em um determinado momento exatamente uma ocorrência daquele campo pode ser nula.)

A Regra de Integridade 2 (integridade referencial) não é obrigatória

A Regra de Integridade 2 Proíbe referências a registros inexistentes (novamente para detalhes, veja o Capítulo 4). Por exemplo, se um determinado registro SP contiver um valor *s* no campo S#, então deverá também existir um registro S com aquele mesmo valor *s* no campo S#. A linguagem SQL definida em [5.5] e [6.1] incluiu um mecanismo geral de *declaração* que permitiria a especificação opcional de restrições tais como a Regra 2 (e restrições mais gerais), mas declarações não estão suportadas no Sistema R.

EXERCÍCIOS

6.1 Defina um esquema para a sua solução do Exercício 3.1 (estrutura relacional para o banco de dados de pessoas e habilidades), usando a SQL DDL.

6.2 A Fig. 6.2 mostra um exemplo de extensão ao banco de dados contendo informações sobre fornecedores (S), peças (P), e projetos (J). Fornecedores, peças e projetos estão identificados univocamente pelo número do fornecedor (S#), número da peça (P#), e número do projeto (J#), respectivamente. Um registro SPJ significa que o fornecedor especificado fornece a peça especificada para o projeto especificado na quantidade especificada (e a combinação S#-P#-J# define de maneira única tal registro). Defina um esquema para este banco de dados usando a SQL DDL. *Nota:* Este banco de dados será usado em numerosos exercícios nos capítulos subsequentes.

S	S#	SNAME	STATUS	CITY	SPJ	S#	P#	J#	QTY
	S1	Smith	20	London		S1	P1	J1	200
	S2	Jones	10	Paris		S1	P1	J4	700
	S3	Blake	30	Paris		S2	P3	J1	400
	S4	Clark	20	London		S2	P3	J2	200
	S5	Adams	30	Athens		S2	P3	J3	200
P	P#	PNAME	COLOR	WEIGHT	CITY	S2	P3	J4	500
	P1	Nut	Red	12	London	S2	P3	J5	600
	P2	Bolt	Green	17	Paris	S2	P3	J6	400
	P3	Screw	Blue	17	Rome	S2	P3	J7	800
	P4	Screw	Red	14	London	S2	P5	J2	100
	P5	Cam	Blue	12	Paris	S3	P3	J1	200
	P6	Cog	Red	19	London	S3	P4	J2	500
J	J#	JNAME	CITY			S4	P6	J3	300
	J1	Sorter	Paris			S4	P6	J7	300
	J2	Punch	Rome			S5	P2	J2	200
	J3	Reader	Athens			S5	P2	J4	100
	J4	Console	Athens			S5	P5	J5	500
	J5	Collator	London			S5	P6	J7	100
	J6	Terminal	Oslo			S5	P6	J2	200
	J7	Tape	London			S5	P1	J4	1000

Fig. 6.2 O banco de dados de fornecedores-peças-projetos.

6.3 Como poderia a SQL DDL ser estendida para incorporar domínios?

REFERÊNCIAS E BIBLIOGRAFIA

Veja também todas as referências do Capítulo 5.

6.1 R. F. Boyce and D. D. Chamberlin. "Using a Structured English Query Language as a Data Definition Facility." IBM research Report RJ1318 (dezembro de 1973).

7

Manipulação de Dados do Sistema R

7.1 INTRODUÇÃO

Neste capítulo vamos considerar os aspectos de manipulação de dados do Sistema R. Em particular, apresentaremos as porções DML da linguagem SQL. A SQL DML opera sobre tabelas básicas e visões, como explicado no Capítulo 5, mas no momento vamos nos preocupar somente com as tabelas básicas; continuaremos também a supor que as operações estão sendo iniciadas e os resultados sendo projetados em um terminal *on-line* (a SQL embutida será discutida no Capítulo 8). Como sempre, todos os exemplos estarão se baseando no banco de dados de fornecedores e peças da Fig. 4.7.

A linguagem definida na referência [5.5] inclui alguns dispositivos da SQL DML que não estão suportados pelo Sistema R. Vamos ignorar esses dispositivos neste capítulo.

Nota: Muitos exemplos, especialmente os últimos, são bastante complexos. O leitor não deve inferir que a SQL DML é que é complexa. Pelo contrário, o ponto é que as operações são tão simples em SQL (e linguagens comparáveis), que exemplos dessas operações tornar-se-iam bastante desinteressantes, não ilustrando a plena potência da linguagem. Naturalmente mostraremos alguns exemplos simples no início.

7.2 OPERAÇÕES DE RECUPERAÇÃO

A operação fundamental em SQL é o *mapeamento*, representado sintaticamente como um bloco SELECT-FROM-WHERE (selecione-de-no qual). Por exemplo, a consulta “Obtenha os números dos fornecedores e os status dos fornecedores de Paris” pode ser representada como se segue.

```
SELECT S#, STATUS
  FROM S
 WHERE CITY = 'PARIS'
```

Resultado:

S#	STATUS
S2	10
S3	30

Deste exemplo podemos verificar que a operação de “mapeamento” é efetivamente a obtenção de um subconjunto horizontal (encontre todas as linhas nas quais CITY = ‘PARIS’) seguido de um subconjunto vertical (extraia S# e STATUS dessas linhas). Em termos algébricos, podemos considerar como sendo um SELECT seguido de um PROJECT, com a exceção de que, como veremos, a obtenção do subconjunto horizontal pode ser consideravelmente mais sofisticada do que o simples SELECT algébrico do Capítulo 3. (Não confunda o SELECT algébrico com o SELECT da SQL.)

Vamos agora ilustrar os principais dispositivos da linguagem de recuperação através de um conjunto de exemplos cuidadosamente desenvolvidos.

7.2.1 Recuperação simples · Obtenha os números de peças de todas as peças fornecidas.

```
SELECT P#
      FROM SP
```

Resultado:

P#
P1
P2
P3
P4
P5
P6

Nós propusemos acima que um mapeamento (SELECT-FROM-WHERE) possa ser visto como um subconjunto horizontal seguido de uma projeção. Neste exemplo o subconjunto horizontal é a tabela toda (não há cláusula WHERE). Falando em projeção, lembramos ao leitor que PROJECT, como formalmente definido (veja Capítulo 3), opera extraíndo colunas especificadas e depois *eliminando linhas duplicatas redundantes* das colunas extraídas (o resultado é uma *relação*, sem linhas duplicadas). Entretanto, a SQL geralmente não elimina duplicatas do resultado de uma instrução SELECT a menos que o usuário o solicite explicitamente por meio da palavra-chave UNIQUE, como no seguinte exemplo.

```
SELECT UNIQUE P#
      FROM SP
```

Resultado:

P#
P1
P2
P3
P4
P5
P6

A justificativa para fazer com que o usuário especifique UNIQUE nesses casos é que (a) a eliminação de duplicatas pode ter uma operação custosa, e (b) freqüentemente os usuários não serão importunados pela presença de duplicatas em suas saídas. Incidentalmente, a mesma filosofia foi adotada pelo INGRES (veja no Capítulo 13).

7.2.2 Recuperação simples – Obtenha todos os detalhes de todos os fornecedores.

```
SELECT *
  FROM S
```

Resultado: Uma cópia de toda a tabela S.

O asterisco resume uma lista ordenada de todos os nomes de campo da tabela FROM (como especificado pelo dicionário do Sistema R no momento em que o SELECT é pré-compilado; nenhum campo novo adicionado subsequentemente à tabela via EXPAND TABLE será incluído). A instrução SELECT mostrada é portanto equivalente a

```
SELECT S#, SNAME, STATUS, CITY
  FROM S
```

7.2.3 Recuperação qualificada – Obtenha os números dos fornecedores de Paris com status > 20

```
SELECT S#
  FROM S
 WHERE CITY = 'PARIS'
   AND STATUS > 20
```

Resultado:

S#
S3

A condição ou *predicado* que segue WHERE pode incluir operadores de comparação =, \neq (não igual), $>$, $>=$, $<$, e $<=$; os operadores Booleanos AND, OR, e NOT; e parêntesis para indicar uma ordem desejada de avaliação.

7.2.4 Recuperação com ordenação – Obtenha os números dos fornecedores e os status dos fornecedores de Paris, em ordem decrescente de status.

```
SELECT S#, STATUS
  FROM S
 WHERE CITY = 'PARIS'
 ORDER BY STATUS DESC
```

Resultado:

S#	STATUS
S3	30
S2	10

Em geral, não há garantia de que o resultado de SELECT esteja em nenhuma sequência específica. Neste caso, entretanto, o usuário especificou que o resultado deveria ser orde-

nado em uma ordem particular antes de ser apresentado. A ordenação pode ser especificada da mesma maneira que em CREATE INDEX (Seção 6.3) – isto é,

```
field-name [order] [,field-name [order]] ...
```

7.2.5 Recuperação de mais de uma tabela – Para cada peça fornecida, obtenha o número da peça e os nomes de todas as cidades que fornecem a peça.

```
SELECT UNIQUE P#,CITY  
FROM SP,S  
WHERE SP.S#=S.S#
```

Resultado:

P#	CITY
P1	London
P1	Paris
P2	London
P2	Paris
P3	London
P4	London
P5	London
P6	London

Este exemplo mostra como uma união é expressa em SQL. O usuário pode nomear diversas tabelas na cláusula FROM, e pode usar os nomes de tabelas como qualificadores nas cláusulas SELECT e WHERE para resolver ambigüidades, se necessário (ou para maior clareza). A cláusula SELECT no exemplo anterior poderia ter sido escrita como

```
SELECT UNIQUE SP.P#, S.CITY
```

se desejado.

Conceitualmente, a instrução SELECT neste exemplo opera da seguinte maneira: (a) O produto Cartesiano de SP e S é formado; (b) as linhas que não satisfazem à “condição de união” SP.S# = S.S# são eliminadas do resultado da etapa (a); (c) as colunas P# e CITY são extraídas do resultado da etapa (b); (d) as linhas duplicadas redundantes são eliminadas do resultado da etapa (c). (Não pretendemos dizer que o Sistema R realmente implemente a consulta desta forma – há maneiras mais eficientes de fazê-lo – mas o leitor deve se convencer de que, sem dúvida, este procedimento produz o resultado desejado. Também o “produto Cartesiano” aqui na verdade se refere a um produto Cartesiano *estendido*, a ser discutido no Capítulo 12, e não à forma simples definida no Capítulo 4.)

7.2.6 Recuperação envolvendo a união de uma tabela com ela mesma – Obtenha todos os pares de números de fornecedores tais que os dois fornecedores estejam localizados na mesma cidade.

```
SELECT FIRST.S#, SECOND.S#  
FROM S FIRST, S SECOND  
WHERE FIRST.CITY = SECOND.CITY  
AND FIRST.S# < SECOND.S#
```

Resultado:

S#	S#
S1	S4
S2	S3

Este exemplo envolve a união da tabela S com ela mesma; a condição de união é que as duas cidades sejam a mesma, e que o primeiro número de fornecedor seja menor do que o segundo. A tabela S aparece duas vezes na cláusula FROM. Para fazer uma distinção entre esses dois aparecimentos, introduzimos os nomes arbitrários FIRST e SECOND, usando-os como qualificadores nas cláusulas SELECT e WHERE. (Há duas razões para a exigência de que o primeiro número de fornecedor seja menor do que o segundo: (a) eliminar os pares da forma (x, x) ; (b) garantir que pelo menos um dos pares $(x, y), (y, x)$ apareça.)

7.2.7 Recuperação usando ANY – Obtenha os nomes dos fornecedores que fornecem a peça P2.

A SQL oferece diversas maneiras para se manusear esta consulta. Uma forma é usando a união:

```
SELECT UNIQUE SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = 'P2'
```

Resultado:

SNAME
Smith
Jones
Blake
Clark

(Note que o UNIQUE é necessário porque a união de S e SP em S# irá em geral conter SNAMEs duplicados.)

Entretanto, este exemplo difere dos dois exemplos anteriores, pois o resultado é inteiramente extraído de uma única tabela, a tabela S – embora seja verdade que têm que ser inspecionadas duas tabelas, S e SP, para determinar o resultado. É portanto possível expressar a consulta da forma

```
SELECT SNAME
FROM   S
WHERE  condition involving P2
```

onde “condition involving P2” representa a condição que tem que ser satisfeita por qualquer fornecedor cujo SNAME for extraído. Qual é essa condição? Basicamente a de que o fornecedor tem que ser um daqueles que fornecem a peça P2 – em outras palavras, de que o valor S# do fornecedor tem que ser um dos valores S# correspondentes a P# ‘P2’

na tabela de embarques. Esta condição pode ser expressa como se segue (repetimos as cláusulas SELECT e FROM para maior clareza).

```
SELECT SNAME
  FROM   S
 WHERE  S# = ANY (SELECT S#
                   FROM   SP
                   WHERE  P# = 'P2')
```

A expressão entre parêntesis é uma *subconsulta*. Neste caso, ela avalia o conjunto de números de fornecedores que corresponde à peça P2 na tabela SP, isto é, o conjunto { 'S1', 'S2', 'S3', 'S4' }. A condição

```
S# = ANY ( { 'S1', 'S2', 'S3', 'S4' } )
```

então avalia como *verdadeiro* se e somente se S# tiver um dos valores S1, S2, S3, S4; dessa forma a consulta completa produz o resultado mostrado anteriormente.

Em geral, o operador = ANY é interpretado como se segue. A condição

```
f = ANY (SELECT F FROM ...)
```

avalia como *verdadeiro* se e somente se o valor f for igual a pelo menos um valor no resultado da avaliação de "SELECT FROM ..." Semelhantemente, a condição

```
f < ANY (SELECT F FROM ...)
```

avalia como *verdadeiro* se e somente se f for menor do que pelo menos um valor no resultado da avaliação de SELECT. Os operadores \leq ANY, $>$ ANY, \geq ANY, e \neq ANY são definidos de forma análoga.

Vamos dar um exemplo do uso de < ANY.

7.2.8 Recuperação usando < ANY – Obtenha os números dos fornecedores cujo valor de status seja menor do que o máximo valor corrente de status na tabela S. (Uma solução mais simples a este problema será dada mais adiante como exemplo 7.3.5.)

```
SELECT  S#
  FROM   S
 WHERE   STATUS < ANY (SELECT STATUS
                         FROM   S)
```

Resultado:

S#
S1
S2
S4

O objetivo ao mostrar este exemplo foi o de alertar o leitor para uma possível armadilha. A cláusula WHERE *não* significa que os status dos fornecedores selecionados sejam menores do que qualquer valor corrente de status, no sentido em que aquela (admitidamente ambígua) expressão seria normalmente entendida em inglês; ao invés disso, ela significa que o status dos fornecedores selecionados é menor do que *algum* valor corrente de status (sendo consequentemente menor do que o máximo corrente). A interpretação intuitiva de

< ANY como “menor do que *qualquer*” é errônea. SOME poderia ser uma palavra chave melhor do que ANY.

Das várias comparações ANY, certamente a mais útil é = ANY. Esta forma pode ser equivalentemente (e mais claramente) escrita como IN, como ilustra o seguinte exemplo.

7.2.9 Recuperação usando IN – Obtenha os nomes dos fornecedores que fornecem a peça P2 (mesmo do exemplo 7.2.7).

```
SELECT SNAME
  FROM S
 WHERE S# IN
       (SELECT S#
        FROM SP
       WHERE P# = 'P2')
```

“IN” e “= ANY” são totalmente intercambiáveis. Cada um deles pode ser visto como o operador de conjuntos \in . Nós geralmente usaremos IN ao invés de = ANY a partir de agora.

7.2.10 Recuperação usando múltiplos níveis de ninho – Obtenha os nomes dos fornecedores que fornecem pelo menos uma peça vermelha.

```
SELECT SNAME
  FROM S
 WHERE S# IN
       (SELECT S#
        FROM SP
       WHERE P# IN
             (SELECT P#
              FROM P
             WHERE COLOR = 'RED'))
```

Resultado:

SNAME
Smith
Jones
Clark

Subconsultas podem conter ninhos em qualquer profundidade.

7.2.11 Recuperação com subconsulta, com referência entre blocos – Obtenha os nomes dos fornecedores que fornecem a peça P2 (mesmo que nos Exemplos 7.2.7 e 7.2.9).

Vamos mostrar outra solução deste problema para ilustrar outro novo ponto.

```
SELECT SNAME
  FROM S
 WHERE 'P2' IN
       (SELECT P#
        FROM SP
       WHERE S# = S.S#)
```

Aqui, na última linha, a referência não qualificada a S# está implicitamente qualificada pelo nome de tabela SP. Para se poder referenciar ao S# do bloco externo dentro do bloco interno, é necessário um qualificador (S) explícito.

7.2.12 Recuperação com subconsulta, com a mesma tabela envolvida nos dois blocos – Obtenha os números dos fornecedores que fornecem pelo menos uma peça fornecida pelo fornecedor S2.

```
SELECT UNIQUE S#
  FROM   SP
 WHERE  P# IN
        (SELECT P#
          FROM   SP
         WHERE  S# = 'S2')
```

Resultado:

S#
S1
S2
S3
S4

Novamente, a referência não qualificada a S# na última linha está implicitamente qualificada pelo nome de tabela SP. De fato, a referência é inteiramente local ao bloco interno; Não tem o mesmo significado que a referência a S# (ou SP.S#) no bloco externo (primeira linha). A questão que surge, é como *podemos* fazer referência ao S# externo no bloco interno, já que a mesma tabela está envolvida nos dois blocos? A resposta é que introduziremos outro nome no bloco externo e o usaremos como qualificador, como ilustra o Exemplo 7.2.13 (compare com o Exemplo 7.2.6).

7.2.13 Recuperação com subconsulta, com referência entre blocos e a mesma tabela envolvida nos dois blocos – Obtenha os números de peças de todas as peças fornecidas por mais de um fornecedor. (Uma solução mais simples a este problema será dado mais adiante, como Exemplo 7.3.7).

```
SELECT UNIQUE P#
  FROM   SP SPX
 WHERE  P# IN
        (SELECT P#
          FROM   SP
         WHERE  S# != SPX.S#)
```

Resultado:

P#
P1
P2
P4
P5

SPX é um nome arbitrário usado para ligar a referência SPX.S# no bloco interno à tabela do bloco externo. A operação de consulta pode ser explicada da seguinte maneira: "De uma linha por vez, digamos SPX, da tabela SP, extraia o valor P# se esse valor P# estiver no conjunto dos valores P# cujo correspondente valor S# não seja aquele na linha SPX – isto é, se aquela peça for fornecida por algum fornecedor diferente do identificado pela linha SPX".

7.2.14 Recuperação usando ALL – Obtenha os nomes dos fornecedores que não fornecem a peça P2.

Novamente há muitas soluções possíveis; a que está mostrada foi escolhida para ilustrar uma comparação ALL (definida de maneira análoga à comparação ANY).

```
SELECT SNAME
FROM   S
WHERE  'P2'  ⊘=ALL
       (SELECT P#
        FROM   SP
        WHERE  S# = S.S#)
```

Resultado:

SNAME
Adams

Em geral, o operador * ALL (onde * é qualquer dos =, ⊘ =, >, ≥, <, ≤) é definido como se segue. A condição

```
f *ALL (SELECT F FROM ...)
```

é avaliada como *verdadeira* se e somente se a comparação "f * V" for avaliada como *verdadeira* para todos os valores de V no resultado da avaliação de "SELECT FROM...". No exemplo acima, portanto, os fornecedores serão selecionados se e somente se o valor 'P2' for não igual a qualquer dos números de peças que eles fornecem. (Como no caso de ANY, há uma armadilha aqui para os incautos.)

Da mesma forma que = ANY pode ser escrito IN, também ⊘ = ALL pode ser escrito como NOT IN.

Algumas vezes é possível que o usuário saiba que uma determinada subconsulta deve retornar exatamente um valor. Nesse caso, = ANY, > ALL, etc podem ser abreviados nas formas não qualificadas =, >, etc. Segue-se um exemplo.

7.2.15 Recuperação com subconsulta e operador de comparação não-qualificado – Obtenha os números dos fornecedores localizados na mesma cidade que o fornecedor S1.

```
SELECT S#
FROM   S
WHERE  CITY =
       (SELECT CITY
        FROM   S
        WHERE  S# = 'S1')
```

Resultado:

S#
S1
S4

7.2.16 Recuperação usando EXISTS – Obtenha os nomes dos fornecedores que fornecem a peça P2 (mesmo que os Exemplos 7.2.7, 7.2.9, e 7.2.11).

```
SELECT SNAME
FROM S
WHERE EXISTS
  (SELECT *
   FROM SP
   WHERE S# = S.S#
     AND P# = 'P2')
```

EXISTS aqui representa um *quantificador existencial*. A expressão “EXISTS (SELECT . . .)” é avaliada como *verdadeira* se e somente se o resultado da avaliação de “SELECT . . .” não for vazio, isto é, se e somente se existir um registro na tabela indicada (a tabela SP no exemplo) satisfazendo à condição WHERE no “SELECT . . .”. Uma forma de pensar a respeito deste exemplo é tomar um SNAME de cada vez e verificar se ele faz com que o teste de existência seja avaliado como *verdadeiro*. O primeiro valor de SNAME na Fig. 4.7 é ‘Smith’; o valor correspondente de S# é ‘S1’; existe algum registro SP com S# igual a ‘S1’ e P# igual a ‘P2’? Se a resposta for sim, ‘Smith’ será um dos valores recuperados. O mesmo deve ser feito para os restantes valores de SNAME.

7.2.17 Recuperação usando NOT EXISTS – Obtenha os nomes dos fornecedores que não fornecem a peça P2 (mesmo que o Exemplo 7.2.14).

```
SELECT SNAME
FROM S
WHERE NOT EXISTS
  (SELECT *
   FROM SP
   WHERE S# = S.S#
     AND P# = 'P2')
```

A consulta pode ser parafraseada como se segue: “Selecione os nomes dos fornecedores para os quais não exista um embarque relacionando-os com a peça P2.”

7.2.18 Recuperação usando NOT EXISTS – Obtenha os nomes dos fornecedores que fornecem todas as peças.

```
SELECT SNAME
FROM S
WHERE NOT EXISTS
  (SELECT *
   FROM P
   WHERE NOT EXISTS
     (SELECT *
      FROM SP
      WHERE S# = S.S#
        AND P# = P.P#))
```

Resultado:

SNAME
Smith

A consulta pode ser parafraseada: "Selecione os nomes dos fornecedores para os quais não existe uma peça que eles não fornecem."

7.2.19 Recuperação usando NOT EXISTS – Obtenha os números dos fornecedores que fornecem pelo menos todas as peças fornecidas pelo fornecedor S2.

Uma forma de se atacar este problema é dividi-lo em consultas menores. Poderemos então primeiramente descobrir o conjunto de números de peças fornecidas pelo fornecedor S2:

```
SELECT P#
  FROM SP
 WHERE S# = 'S2'
```

Resultado:

P#
P1
P2

Usando CREATE TABLE e INSERT (discutidos na Seção 7.4), é possível guardar-se esse resultado em uma tabela temporária, digamos TEMP, no banco de dados. Podemos então prosseguir para encontrar o conjunto de números de fornecedores que fornecem todas as peças listadas em TEMP (compare com o Exemplo 7.2.18):

```
SELECT UNIQUE S#
  FROM SP SPX
 WHERE NOT EXISTS
   (SELECT *
      FROM TEMP
     WHERE NOT EXISTS
       (SELECT *
          FROM SP
         WHERE S# = SPX.S#
        AND P# = TEMP.P#))
```

Resultado:

S#
S1
S2

[Note, entretanto, que temos que introduzir outro nome (SPX), pois aqui estamos extraíndo valores S# da tabela SP, e não valores SNAME da tabela S.] A tabela TEMP pode agora ser destruída.

Também é possível expressar toda a consulta como um único SELECT, eliminando a necessidade de TEMP:

```

SELECT UNIQUE S#
FROM   SP SPX
WHERE  NOT EXISTS
       (SELECT *
        FROM   SP SPY
        WHERE  S# = 'S2'
        AND   NOT EXISTS
               (SELECT *
                FROM   SP
                WHERE  S# = SPX.S#
                AND   P# = SPY.P#))

```

7.2.20 Recuperação usando UNION – Obtenha os números das peças que: ou pesam mais do que 18 quilos, ou são correntemente supridas pelo fornecedor S2 (ou ambos).

```

SELECT P#
FROM   P
WHERE  WEIGHT > 18

UNION

SELECT P#
FROM   SP
WHERE  S# = 'S2'

```

Resultado:

P#
P1
P2
P6

UNION é o operador de união da teoria tradicional de conjuntos; em outras palavras, A UNION B (onde A e B são conjuntos) é o conjunto de todos os objetos x tais que x é um membro de A ou x é um membro de B (ou ambos). As duplicações redundantes são sempre eliminadas do resultado de um UNION.

7.2.21 Recuperação de valores calculados – Para todas as peças, obtenha o número da peça e o peso da peça em gramas. Os pesos na tabela P estão em libras.

```

SELECT P#, WEIGHT * 454
FROM   P

```

Resultado:

P#	
P1	5448
P2	7718
P3	7718
P4	6356
P5	5448
P6	8626

A cláusula SELECT (e a cláusula WHERE) pode incluir expressões aritméticas envolvendo campos ou nomes de campos singelos.

7.2.22 Recuperação usando NULL – Suponhamos como exemplo que o fornecedor S5 possua um status nulo, ao invés de 30. Obtenha os números dos fornecedores cujo status seja maior do que 25.

```
SELECT S#
  FROM S
 WHERE STATUS > 25
```

Resultado:

S#
S3

O fornecedor S5 não se qualifica. Quando um valor nulo é comparado com algum outro valor na avaliação de um predicado, não importando que o operador de comparação esteja envolvido, o resultado da comparação é *never verdadeiro*, mesmo que o outro valor seja também nulo. Em outras palavras, nenhuma das seguintes comparações resulta como *verdadeira*¹:

```
null > 25
null < 25
null = 25
null ≠ 25
null = null
```

Um predicado especial, na forma “field IS [NOT] NULL”, existe para se testar se um determinado campo é ou não nulo. Por exemplo,

```
SELECT S#
  FROM S
 WHERE STATUS > 25
 OR   STATUS IS NULL
```

Resultado:

S#
S3
S5

Chegamos ao fim dos nossos exemplos de recuperação. Ficou claro que a linguagem de recuperação é muito poderosa. Ela é, de fato, *relacionalmente completa* [12.1]. Ser relationalmente completa é uma medida básica da potência seletiva de uma linguagem. Uma linguagem é dita ser relationalmente completa se qualquer relação derivável das relações

¹

Na realidade, como indicado no Capítulo 6, todos resultam como *desconhecidos*. Os predicados são avaliados usando-se os três valores lógicos (*verdadeiro*, *falso*, *desconhecido*). A cláusula WHERE selecionará os registros para os quais a avaliação resultar como *verdadeira*, isto é, nem *falsa* nem *desconhecida*.

dadas (isto é, o banco de dados) por meio de uma expressão do *cálculo relacional* (veja Capítulo 13) puder ser recuperada usando-se essa linguagem. O que torna a SQL (e linguagens semelhantes) tão potente é a economia com que essa forma completa é conseguida: qualquer relação derivável pode ser recuperada usando-se *uma única instrução* da linguagem. De fato, o termo “relacionalmente completa” freqüentemente inclui esta restrição adicional (ou seja, que qualquer relação derivável possa ser recuperada com uma única instrução), e neste livro nós geralmente usaremos o termo neste sentido mais exigente. A maior parte das linguagens relacionais a nível de conjunto discutidas neste livro tem, pelo menos, essa potência; linguagens de nível mais baixo, relacionais ou não, certamente não têm. O que “relacionalmente completa” significa para o usuário é que, falando de forma *muito* superficial, se a informação desejada encontra-se no banco de dados, então ela pode ser recuperada por meio de uma única solicitação auto-suficiente (mas veja a Seção 7.3). Em linguagens de nível mais baixo, o usuário tem que escrever procedimentos bastante complicados para responder a todas as questões, exceto algumas muito simples. É verdade que a maioria das consultas é bastante simples na prática; mas nós sabemos, da teoria esboçada, que o usuário *pode* levantar questões arbitrariamente complexas, se necessário.

7.3 FUNÇÕES INTEGRADAS

Embora “completa”, como acabamos de explicar, a linguagem de recuperação até agora descrita é ainda inadequada para muitos problemas práticos. Por exemplo, mesmo uma consulta tão simples como “Quantos fornecedores existem?” não pode ser expressa usando-se somente as construções dadas na Seção 7.2. Por isso a SQL oferece uma quantidade de *funções integradas* para ampliar sua potência básica de recuperação. As funções correntemente suportadas são COUNT, SUM, AVG, MAX, e MIN. Fora o caso especial de “COUNT(*)” (veja abaixo), cada uma dessas funções opera sobre a coleção de valores em uma coluna de alguma tabela, produzindo um resultado como a seguir:

- COUNT: quantidade de valores
- SUM: soma dos valores
- AVG: média dos valores
- MAX: maior valor
- MIN: menor valor

Para SUM e AVG a coluna operada tem que conter valores numéricos. De forma geral, o argumento da função pode opcionalmente ser precedida da palavra-chave UNIQUE para indicar que os valores duplicados redundantes devem ser eliminados antes da aplicação da função. UNIQUE *tem* que ser especificado para COUNT; entretanto, é fornecida a função especial COUNT(*) para contar todas as linhas de uma tabela sem nenhuma eliminação de duplicatas.

Quaisquer valores nulos no argumento são sempre eliminados antes da aplicação da função, tenha sido ou não especificado UNIQUE, *exceto* no caso de COUNT(*), quando as linhas são contadas mesmo se elas forem todas nulas (isto é, consistindo inteiramente de campos nulos).

Se o argumento for um conjunto vazio, COUNT retorna um resultado zero; todas as outras funções retornam nulo.

7.3.1 Função na cláusula SELECT – Obtenha a quantidade total de fornecedores.

```
SELECT COUNT (*)
FROM S
```

Resultado:

5

7.3.2 Função na cláusula SELECT – Obtenha a quantidade total de fornecedores correntemente fornecendo peças.

```
SELECT COUNT (UNIQUE S#)
FROM SP
```

Resultado:

4

7.3.3 Função na cláusula SELECT, com um predicado – Obtenha a quantidade total de embarques da peça P2.

```
SELECT COUNT(*)
FROM SP
WHERE P# = 'P2'
```

Resultado:

4

7.3.4 Função na cláusula SELECT, com um predicado – Obtenha a quantidade total de peças P2 fornecidas.

```
SELECT SUM(QTY)
FROM SP
WHERE P# = 'P2'
```

Resultado:

1000

7.3.5 Função em uma subconsulta – Obtenha os números dos fornecedores cujo valor de status seja menor do que o máximo valor corrente de status na tabela S. (Mesmo do Exemplo 7.2.8).

```
SELECT S#
FROM S
WHERE STATUS <
      (SELECT MAX(STATUS)
       FROM S)
```

7.3.6 Uso de GROUP BY – Para cada peça fornecida, obtenha o número da peça e a quantidade total fornecida daquela peça.

```
SELECT P#, SUM(QTY)
FROM   SP
GROUP  BY P#
```

Resultado:

P#	
P1	600
P2	1000
P3	400
P4	500
P5	500
P6	100

O operador GROUP BY conceitualmente rearranja a tabela FROM em partições ou *grupos*, de tal forma que dentro de qualquer dos grupos todas as linhas tenham o mesmo valor no campo do GROUP BY. (Isto, naturalmente, não significa que a tabela seja fisicamente rearranjada no banco de dados). No exemplo, a tabela SP está agrupada de forma a que o primeiro grupo contenha as linhas da peça P1, o segundo grupo contenha as linhas da peça P2, e assim sucessivamente. A cláusula SELECT é então aplicada a cada grupo da tabela particionada (ao invés de a cada linha da tabela original). Cada expressão da cláusula SELECT tem que ter *um só valor* para cada grupo; isto é, pode ser o próprio campo de GROUP BY, ou uma função tal como SUM, que opera sobre todos os valores de um determinado campo dentro de um grupo e reduz aqueles valores a um único.

Uma tabela pode ser agrupada por qualquer combinação dos seus campos.

7.3.7 Uso de GROUP BY com HAVING – Obtenha os números de peças para todas aquelas fornecidas por mais de um fornecedor (mesmo que o exemplo 7.2.13).

```
SELECT P#
FROM   SP
GROUP  BY P#
HAVING COUNT(*) > 1
```

HAVING funciona para grupos da mesma forma que WHERE funciona para linhas. (Se HAVING for especificado, GROUP BY também tem que ser especificado.) As expressões na cláusula HAVING têm que ter um só valor para cada grupo.

Note que a cláusula SELECT não requer mais a especificação UNIQUE (em contraste com o exemplo 7.2.13).

7.3.8 Um exemplo abrangente – Para todas as peças cuja quantidade total fornecida for maior do que 300 (excluindo do total todos os embarques cuja quantidade é igual ou inferior a 200), obtenha o número de peça e a quantidade máxima fornecida daquela peça; ordene o resultado de forma descendente por número de peça dentro das quantidades máximas.

```

SELECT P#, MAX(QTY)
FROM SP
WHERE QTY > 200
GROUP BY P#
HAVING SUM(QTY) > 300
ORDER BY 2, P# DESC

```

Resultado:

P#	
P1	300
P5	400
P3	400
P2	400

As cláusulas FROM, WHERE, GROUP BY e HAVING estão aplicadas na ordem devida. Assim, podemos imaginar a formação do resultado do exemplo como se segue.

1. (FROM) É feita uma cópia da tabela SP.
2. (WHERE) São eliminadas as linhas que não satisfazem a “QTY > 200”.
3. (GROUP BY) As linhas restantes são grupadas por P#.
4. (HAVING) São eliminados os grupos que não satisfazem a “SUM (QTY) > 300”.
5. (SELECT) Os números de peças e quantidades máximas são extraídos dos grupos restantes.
6. (ORDER BY) A tabela então formada recebe a ordenação especificada. No exemplo, como nós queremos ordenar o resultado (em peça) por valores do segundo campo do resultado e aquele campo não possui um nome de campo, usamos sua posição ordinal dentro do resultado (2) na especificação de ORDER BY.

7.4 OPERAÇÕES DE ATUALIZAÇÃO

A SQL DML inclui três operações de atualização: UPDATE (modifique), INSERT, e DELETE.

7.4.1 Atualização de um só registro – Mude a cor da peça P2 para amarela, aumente seu peso de 5, e ajuste a sua cidade para “desconhecida” (NULL).

```

UPDATE P
SET COLOR = 'YELLOW',
WEIGHT = WEIGHT + 5,
CITY = NULL
WHERE P# = 'P2'

```

Dentro de uma cláusula SET, qualquer referência a um campo à direita de um sinal de igual se refere ao valor do campo antes que tenha ocorrido qualquer atualização.

7.4.2 Atualização de múltiplos registros – Dobre o status de todos os fornecedores de Londres.

```
UPDATE S
SET    STATUS = 2 * STATUS
WHERE   CITY = 'LONDON'
```

7.4.3 Atualização com subconsulta — Zere as quantidades de todos os fornecedores de Londres.

```
UPDATE SP
SET    QTY = 0
WHERE  'LONDON' =
(SELECT CITY
 FROM   S
 WHERE  S# = SP.S#)
```

7.4.4 Atualizações de múltiplas tabelas — Mude o número do fornecedor S2 para S9.

```
UPDATE S
SET    S# = 'S9'
WHERE  S# = 'S2'
```

```
UPDATE SP
SET    S# = 'S9'
WHERE  S# = 'S2'
```

Não é possível atualizar mais do que uma tabela com uma única instrução UPDATE; colo-
cando de outra forma, a cláusula UPDATE tem que especificar exatamente uma tabela.
Neste exemplo, portanto, nós temos um *problema de integridade*: o banco de dados torna-
se inconsistente após o primeiro UPDATE, e assim permanece até o segundo. Reverter a seqüência dos dois UPDATE não resolve o problema. De fato, neste exemplo específico,
foi temporariamente violada a Regra de Integridade 2, pois é a chave primária que está
sendo modificada. Por isso alguns sistemas (não o Sistema R) proibem atualizações da
chave primária (modificar a chave primária nesses sistemas envolve uma remoção seguida
de uma inserção). Entretanto, a atualização de múltiplas tabelas — isto é, execução de
operações interligadas sobre múltiplas tabelas — pode de qualquer forma causar problemas
de inconsistência, mesmo que a Regra de Integridade 2 em si não esteja envolvida.

7.4.5 Inserção de registro único — Adicione a peça P7 (nome 'WASHER', cor 'GREY', peso 2, cidade 'ATHENS') à tabela P.

```
INSERT INTO P:
  <'P7','WASHER','GREY',2,'ATHENS'>
```

(Note que a operação confia na ordem de colunas da esquerda para a direita dentro da
tabela.) Também é possível inserir um novo registro do qual não se conheça todos os
campos. Por exemplo,

```
INSERT INTO P (P#, WEIGHT, PNAME):
  <'P7',2,'WASHER'>
```

É criado um novo registro P com as especificações dadas de número, peso e nome, e com

valores nulos para cor e cidade. É claro que esses campos não podem ser definidos com a opção NONULL. A ordem em que estes campos estão nomeados neste formato de INSERT não precisa ser a sua mesma ordem dentro da tabela.

INSERT pode ser usado para se inserir um único registro, como no exemplo acima, ou um conjunto completo de registros resultantes de uma consulta interna, como no próximo exemplo.

7.4.6 Inserção de múltiplos registros – A tabela TEMP possui uma coluna, chamada P#. Dê entrada em TEMP dos números de todas as peças fornecidas pelo fornecedor S2 (veja Exemplo 7.2.19).

```
INSERT INTO TEMP:  
    SELECT P#  
      FROM SP  
     WHERE S# = 'S2'
```

Não é necessário que a tabela objetivo esteja inicialmente vazia para uma inserção de múltiplos registros com INSERT, como estaria no Exemplo 7.2.19.

7.4.7 Remoção de registro único – Remova o fornecedor S1.

```
DELETE S  
  WHERE S# = 'S1'
```

Se a tabela SP tiver algum embarque corrente para o fornecedor S1, este DELETE causará violação da Regra de Integridade 2.

7.4.8 Remoção de múltiplos registros – Remova todos os embarques.

```
DELETE SP
```

SP continua sendo uma tabela conhecida, mas agora está vazia.

7.4.9 Remoção de múltiplos registros, múltiplas tabelas – Remova todos os embarques dos fornecedores de Londres e também os fornecedores envolvidos.

```
DELETE SP  
  WHERE 'LONDON' =  
        (SELECT CITY  
         FROM S  
        WHERE S# = SP.S#)
```

```
DELETE S  
  WHERE CITY = 'LONDON'
```

7.5 DICIONÁRIO DO SISTEMA R

Lembremo-nos do Capítulo 1 que o dicionário é o componente do sistema que contém “dados sobre dados” – isto é, descrições dos outros objetos no sistema. Em um sistema

relacional, os objetos a serem descritos são, naturalmente, relações, atributos e assim por diante. Se essas próprias descrições forem apresentadas aos usuários na forma de relações – em outras palavras, se o usuário enxergar o próprio dicionário como uma coleção de relações – então, em princípio, a mesma sublinguagem de dados usada para os demais dados pode ser usada para o dicionário, uma bela simplificação conceitual.

Vamos agora considerar explicitamente o Sistema R. Neste, o dicionário é representado para o usuário como um conjunto de relações. Vamos mencionar somente duas dessas relações aqui, TABLES e COLUMNS (não são os nomes reais do Sistema R). TABLES contém uma linha para cada tabela definida no sistema, dando TNAME (nome da tabela; esta é a chave primária), CREATOR (identificação do CRIADOR desta tabela), NCOLS (número de colunas nesta tabela), e outras informações. COLUMNS contém uma linha para cada coluna de cada tabela definida no sistema, dando TNAME e CNAME (nome da tabela e nome da coluna; esta combinação é a chave primária), COLTYPE (tipo de dado desta coluna, por exemplo, caractere ou numérico), LENGTH (número de bytes), e outras informações.

A SQL DML pode ser usada para interrogar essas tabelas do dicionário. Por exemplo, a consulta

```
SELECT TNAME  
FROM   COLUMNS  
WHERE  CNAME = 'S#'
```

irá resultar em uma lista de nomes de todas as tabelas que possuem uma coluna chamada S#. Semelhantemente, a consulta

```
SELECT CNAME  
FROM   COLUMNS  
WHERE  TNAME = 'S'
```

fornecerá uma lista de todos os nomes de colunas da tabela S. Assim, o usuário que desejar obter informação referente a fornecedores mas que inicialmente só sabe que os fornecedores são identificados por um atributo chamado S#, poderia usar as duas consultas acima ao dicionário como preliminares à consulta dos dados, tal como

```
SELECT S#, STATUS, CITY  
FROM   S
```

Deve ficar claro que o sistema é crucialmente dependente da correção das tabelas do dicionário; em particular, é preciso se confiar que aquelas tabelas estejam mutuamente consistentes. Por exemplo, cada TNAME mencionado em COLUMNS tem que aparecer também em TABLES. Por isso seria muito arriscado permitir que os usuários executassem operações de INSERT, DELETE, ou UPDATE não controlados no dicionário, usando a SQL DML comum. Ao invés disso, os usuários normalmente usam operações DDL “amarra-das”, tais como CREATE TABLE. CREATE TABLE, por exemplo, cria uma entrada em TABLES para a nova tabela, e também uma entrada em COLUMNS para cada linha daquela tabela, garantindo que TABLES e COLUMNS estarão consistentes no final da operação. É possível o uso direto de INSERT, DELETE, e UPDATE operando sobre o dicionário, mas somente se o usuário possuir “autoridade de DBA”, e mesmo assim em circunstâncias muito especiais.

Finalmente, as operações DDL CREATE TABLE, DROP TABLE etc. não são normalmente aplicáveis aos objetos no dicionário. Consideremos CREATE TABLE como exemplo. A função de CREATE TABLE é construir uma nova tabela (obtendo o espaço de armazenamento necessário) e, como já vimos, construir as definições apropriadas (entradas no dicionário) para aquela nova tabela. Não é necessário emitir CREATE TABLE para as tabelas do dicionário, pois essas tabelas já existem; elas foram construídas quando o sistema foi inicializado, e suas definições já estão no dicionário (elas estão “ligadas por fios” ao sistema). Por oportuno, a tabela TABLES já contém uma entrada para cada tabela do dicionário – inclusive a própria TABLES – no momento que o sistema é iniciado.

7.6 DISCUSSÃO

Para concluir este capítulo, apresentaremos uma breve discussão sobre as vantagens da SQL DML quando comparada a DMLs de níveis mais baixos (como as DMLs de um registro por vez apresentadas no Capítulo 3). Muitos dos itens que se seguem aplicam-se igualmente a outras DMLs relacionais de nível de conjunto, como veremos em capítulos posteriores.

- Simplicidade

Muitos problemas podem ser expressos em SQL mais fácil e consistentemente do que em linguagens de nível mais baixo. Por sua vez, simplicidade significa maior produtividade. No caso da SQL embutida, isto se aplica tanto à pessoa que produz o programa original quanto às que o irão manter.

- Ser completa

A linguagem é relationalmente completa, como visto na Seção 7.2. Isto significa que, para uma grande classe de consultas, o usuário não precisará apelar para *loops* ou desvios. (Esta afirmativa não é totalmente verdadeira para a SQL embutida; os programadores de aplicação terão freqüentemente que usar alguma forma de estrutura em *loop*. Entretanto, esses *loops* usualmente cairão em uma forma padronizada. Veja o capítulo 8.)

- Ser não-procedural

Linguagens como a SQL DML são freqüentemente descritas como “não-procedurais”. Uma instrução SELECT, por exemplo, especifica somente que dado é desejado, não o procedimento para a obtenção do dado. Colocando de outra forma, tal instrução é uma *instrução de intenção* da parte do usuário, em alto nível. Isto é significativo, pois significa que a implementação é capaz de “*pescar*” a intenção do usuário – isto é, pode entender o que o usuário está tentando fazer, em um nível razoavelmente alto. A captura da intenção do usuário, por sua vez, torna a otimização da pesquisa uma proposição prática. Uma otimização de pesquisa seria muito difícil de ser obtida em DML de nível mais baixo.

A captura da intenção também é importante na implementação de outros aspectos do sistema, tal como verificação de autorização.

- Independência de dados

As instruções da SQL DML não incluem qualquer referência a caminhos explícitos de acesso, como índices ou seqüência física. (Eles podem assumir seqüência lógica.) Portanto, a SQL DML proporciona total independência “física” de dados [7.8] – isto é, independência da forma como os dados estão armazenados fisicamente. Como nós veremos

no Capítulo 9, ela também proporciona um certo grau de independência “lógica” de dados — isto é, independência quanto à forma como os dados estão logicamente estruturados. (DMLs a nível de registro também poderiam proporcionar independência física de dados teoricamente, mas na prática tenderam a não fazê-lo.)

- Facilidade de extensão

A Seção 7.3 mostrou que a potência de recuperação da linguagem básica pode ser facilmente estendida por meio das funções integradas. Alguns sistemas (por exemplo, PRTV [12.5]) permitem que o usuário defina suas próprias funções integradas.

- Suporte para linguagens de nível mais alto

Não estamos tentando dizer que todos os usuários devam usar a SQL ou algo parecido. Pelo contrário, é freqüentemente preferível fornecer uma série de linguagens com fins específicos, sendo cada uma projetada para alguma área particular de aplicação e suportando a terminologia e as operações específicas daquela área. Sem dúvida, alguns usuários “eventuais” podem não dispor de nenhuma linguagem formal para ter acesso ao sistema além das suas linguagens naturais não restritas [7.3—7.5]. De qualquer forma, linguagens como a SQL proporcionam um conjunto de dispositivos que serão requeridos de alguma forma em todas essas linguagens mais elevadas. A SQL, ou alguma outra linguagem de potência comparável, pode portanto ser convenientemente usada como um objetivo comum para aquelas linguagens mais elevadas — isto é, como uma etapa intermediária na tradução daquelas linguagens para a linguagem da máquina básica.

EXERCÍCIOS

Todas as questões desta seção estão baseadas no banco de dados fornecedores-peças-projetos (veja exercícios do Capítulo 6). Alguns valores de dados de exemplo estão mostrados na Fig. 6.2. Você pode encontrar utilidade em interpretar as questões em termos daqueles dados de exemplo; o processo de obter o resultado à mão pode lhe trazer melhor compreensão sobre como formular uma instrução apropriada da SQL. Por conveniência, estamos repetindo aqui os nomes de tabelas e os nomes de campos. As chaves primárias estão mostradas sublinhadas.

```
S      (S#, SNAME, STATUS, CITY)
P      (P#, PNAME, COLOR, WEIGHT, CITY)
J      (J#, JNAME, CITY)
SPJ    (S#, P#, J#, QTY)
```

As questões estão em uma apropriada ordem crescente de complexidade. Você talvez não queira abordá-las todas, pelo menos na primeira leitura; um exemplo representativo está todo numerado com números pares, exercícios 7.2 a 7.20, inclusive, mais todos os exercícios de 7.30 até o final. Os de números 7.21 a 7.28 são bastante difíceis.

Como um exercício adicional, recomendamos fortemente que tente converter as respostas fornecidas para algumas das questões mais complexas de volta para o inglês.

- 7.1 Obtenha todos os detalhes de todos os projetos.
- 7.2 Obtenha todos os detalhes de todos os projetos em Londres.
- 7.3 Obtenha os números das peças de forma a que nenhuma outra peça tenha um valor menor de peso.
- 7.4 Obtenha os valores S# dos fornecedores que fornecem para o projeto J1.
- 7.5 Obtenha os valores de S# para os fornecedores que fornecem a peça P1 para o projeto J1.
- 7.6 Obtenha os valores JNAME dos projetos que são supridos pelo fornecedor S1.

- 7.7 Obtenha os valores de COLOR das peças fornecidas pelo fornecedor S1.
- 7.8 Obtenha os valores de S# dos fornecedores que fornecem tanto para o projeto J1 como para o J2.
- 7.9 Obtenha os valores de S# dos fornecedores que fornecem uma peça vermelha para o projeto J1.
- 7.10 Obtenha os valores de P# de todas as peças fornecidas para qualquer projeto em Londres.
- 7.11 Obtenha os valores S# dos fornecedores que fornecem uma peça vermelha para um projeto em Londres ou Paris.
- 7.12 Obtenha os valores de P# das peças fornecidas para qualquer projeto por um fornecedor na mesma cidade.
- 7.13 Obtenha os valores de P# das peças fornecidas para qualquer projeto em Londres por um fornecedor de Londres.
- 7.14 Obtenha os valores de J# dos projetos supridos por pelo menos um fornecedor não pertencente à mesma cidade.
- 7.15 Obtenha os valores de J# dos projetos que não recebem suprimento de qualquer peça vermelha de nenhum fornecedor de Londres.
- 7.16 Obtenha os valores de S# dos fornecedores que fornecem pelo menos uma peça fornecida por pelo menos um fornecedor que fornece pelo menos uma peça vermelha.
- 7.17 Obtenha os valores de J# dos projetos que usam pelo menos uma peça disponível do fornecedor S1.
- 7.18 Obtenha todos os valores de CITY tais que um fornecedor na primeira cidade forneça suprimento para um projeto na segunda cidade.
- 7.19 Obtenha todas as triplas < CITY, P#, CITY > tais que um fornecedor na primeira cidade forneça a peça especificada a um projeto na segunda cidade.
- 7.20 Repita o Exercício 7.19, mas não recupere as triplas em que os dois valores CITY forem o mesmo.
- 7.21 Obtenha os valores de S# dos fornecedores que fornecem a mesma peça para todos os projetos.
- 7.22 Obtenha os valores de J# dos projetos inteiramente supridos pelo fornecedor S1.
- 7.23 Obtenha os valores de P# das peças fornecidas para todos os projetos em Londres.
- 7.24 Obtenha os valores J# dos projetos supridos com pelo menos todas as peças fornecidas pelo fornecedor S1.
- 7.25 Obtenha os valores de J# dos projetos que só utilizam peças que estão disponíveis no fornecedor S1.
- 7.26 Obtenha os valores de J# dos projetos supridos pelo fornecedor S1 com todas as peças que o fornecedor S1 fornece.
- 7.27 Obtenha os valores de J# dos projetos que obtêm pelo menos alguma das várias peças que usam do fornecedor S1.
- 7.28 Obtenha os valores de J# dos projetos supridos por todos os fornecedores que fornecem alguma peça vermelha.
- 7.29 Mude o nome do projeto J6 para 'VIDEO'.
- 7.30 Mude as cores de todas as peças vermelhas para alaranjado.
- 7.31 Remova todas as peças vermelhas e os correspondentes registros SPJ.
- 7.32 Obtenha a quantidade total de projetos supridos pelo fornecedor S3.
- 7.33 Obtenha a quantidade total de peças fornecidas pelo fornecedor S1.
- 7.34 Para cada peça sendo fornecida a um projeto, obtenha o número da peça, o número do projeto e a quantidade total correspondente.

REFERÊNCIAS E BIBLIOGRAFIA

- 7.1 Relational Software, Inc. ORACLE Introduction – Version 1.3 (1978, 1979). Disponível de RSI, 3000 Sand Hill Road, Menlo Park, California 94025.
- A linguagem SEQUEL/2 [5.5] é usada essencialmente sem modificações como interface do usuário para ORACLE.
- 7.2 Honeywell Information Systems, Inc. Series 60 (Level 68) MULTICS Logical Inquiry and Update System (LINUS). Reference Manual, Preliminary Edition (outubro de 1978).
- LINUS é um sistema de acesso a banco de dados e geração de relatórios do Honeywell MULTICS Relational Data Store (MRDS). Inclui uma “selection language” interativa chamada LILA (Linus Language) que é semelhante à instrução SELECT da SQL. Oferece também funções de atualização que são bem menos parecidas com a SQL.
- 7.3 E. F. Codd. “Seven Steps to Rendezvous with the Casual User.” In *Data Base Management* (eds., Klimbie and Koffeman). Proc. IFIP TC-2 Working Conference on Data Base Management Systems (abril de 1974). North-Holland (1974).
- 7.4 E. F. Codd. “How About Recently?” (English Dialog with Relational Data Bases Using RENDEZVOUS Version 1). In *Databases: Improving Usability and Responsiveness* (ed., B. Schneiderman). New York: Academic Press (1978).
- 7.5 E. F. Codd, R. S. Arnold, J. M. Cadiou, C. L. Chang, and N. Roussopoulos. “RENDEZVOUS Version 1: An Experimental English-Language Query Formulation System for Casual Users of Relational Data Bases.” IBM Research Report RJ12144 (janeiro de 1978).
- Estes artigos [7.3–7.5] dedicam-se a um sistema chamado RENDEZVOUS que permite usuários “eventuais” *on-line* – isto é, usuários sem conhecimentos sobre computadores, programação, ou linguagens artificiais – façam consultas sobre um banco de dados relacional, usando somente sua linguagem natural não restrita. Onde necessário, o sistema interroga o usuário sobre sua consulta, até ser capaz de sintetizar uma representação interna da consulta sob a forma de uma expressão em uma linguagem chamada DEDUCE [7.6] (uma linguagem em um nível comparável ao da SQL), que então a executa. A conversação entre o usuário e o sistema inclui o uso de interrogações de múltipla escolha (seleção de menu) como alternativa suplementar e uma confirmação precisa na linguagem natural do usuário da consulta sobre o que o sistema entendeu dela antes que qualquer dado seja recuperado.
- 7.6 C. L. Chang. “DEDUCE – A Deductive Query Language for Relational Data Bases.” In *Pattern Recognition and Artificial Intelligence* (ed., C. H. Chen). New York: Academic Press (1976).
- 7.7 F. Antonacci, P. Dell’Orco, and V. N. Spadavecchia. “AQL: An APL-Based System for Accessing and Manipulating Data in a Relational Database System.” Proc. APL 76 Conference, Ottawa (setembro de 1976). Disponível da ACM.
- Descreve uma linguagem de consulta parecida com a SQL implementada como uma extensão ao APL.
- 7.8 C. J. Date and P. Hopewell. “Storage Structure and Physical Data Independence.” Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.
- Uma investigação sobre possíveis estruturas de armazenamento para suportar um banco de dados relacional. Uma determinada estrutura de armazenamento é dita ser uma “representação conforme” de uma dada estrutura relacional se existir uma relação de mapeamento de 1-1 entre as duas. São dados diversos exemplos de representações conformes e não conformes, sendo demonstrado por um exemplo que (a) A classe de representações conformes é muito grande, e (b) É difícil implementar mesmo operações relacionais muito simples em termos de representações não conformes. É por isso recomendado que, na prática, só sejam consideradas representações conformes.

8 SQL Embutida

8.1 INTRODUÇÃO

Os Capítulos 6 e 7 discorreram sobre a SQL DML e (partes) da SQL DDL, mostrando como as instruções SQL podem ser usadas em um ambiente interativo. Neste capítulo vamos dirigir nossa atenção para o ambiente de programação, estudando a SQL *embutida*, isto é, a SQL usada pelo programador de aplicação. Como base para nossas idéias, vamos supor que a linguagem básica de programação é o PL/I; em sua maior parte, as colocações servem para o COBOL com alterações muito pequenas.

Comecemos com alguns detalhes preliminares, que estão todos mostrados no fragmento de programa da Fig. 8.1. Primeiro, as instruções da SQL embutida são prefixadas por um sinal \$ (para que possam ser facilmente reconhecidas pelo pré-compilador RDS), e terminadas, como as instruções comuns de PL/I, por um ponto e vírgula (linhas 1, 2, 3, e 8-11). Segundo, uma instrução executável de SQL pode aparecer (a partir de agora normalmente deixaremos de usar o termo “embutida”) onde possam aparecer instruções executáveis de PL/I (linhas 8-11). Terceiro, as instruções SQL podem incluir referências a variáveis do PL/I — isto é, podem incluir os nomes de variáveis ordinárias do PL/I — mas todas essas referências têm que ser prefixadas pelo sinal \$ (linhas 9 e 11), e essas variáveis têm que ser “\$-declared” (isto é, definidas por uma instrução \$DCL ao invés de uma declaração ordinária do PL/I — linhas 1, 2 e 3).

Cabem os seguintes comentários adicionais.

1. A instrução \$SELECT inclui uma cláusula INTO (linha 9) especificando variáveis às quais são atribuídos valores recuperados do banco de dados.

2. Variáveis declaradas com \$ são referenciadas com \$ dentro das instruções SQL (linhas 9 e 11), mas normalmente referenciadas nas instruções PL/I (linha 12). “\$-referências” podem ser usadas para especificar variáveis que devam receber os valores (cláusula INTO em SELECT), comparandos (cláusula WHERE em SELECT, UPDATE e DELETE), novos valores de campos (lista de dados em INSERT), e valores para atualização (cláusula SET em UPDATE).

```

1      $DCL GIVENS# CHAR(5);
2      $DCL RANK FIXED BIN(15);
3      $DCL CITY CHAR(15);
4      DCL ALPHA ... ;
5      DCL BETA ... ;
6      .....
7      IF ALPHA > BETA THEN
8      GETSC:
9          $SELECT STATUS, CITY
10         INTO $RANK, $CITY
11         FROM S
12         WHERE S# = $GIVENS#;
13      .....
14      PUT SKIP LIST (RANK, CITY);

```

Fig. 8.1 Fragmento de um programa PL/I com SQL embutida.

3. As variáveis declaradas com \$ devem ter um tipo de dado PL/I compatível com o tipo de dado do Sistema R dos campos com os quais eles são comparados (linha 11), ou dos quais recebem dados, ou aos quais fornecem dados (linha 9). (O tipo de dado de um campo do Sistema R é especificado na CREATE TABLE ou EXPAND TABLE que define aquele campo.) A compatibilidade de tipo de dado é definida assim: (a) O tipo de dado CHAR do Sistema R é compatível com o tipo de dado CHAR do PL/I, independentemente do comprimento e da especificação de VAR em qualquer dos casos; (b) os tipos de dados numéricos do Sistema R (INTEGER, SMALLINT, FLOAT) são compatíveis com cada um dos tipos de dados PL/I FIXED BIN (15), FIXED BIN (31), FLOAT BIN (53). Se houver perda de dados na transferência (de ou para o banco de dados) por ser o campo objetivo pequeno, será retornada ao programa uma indicação de erro.

4. Variáveis PL/I e campos do banco de dados podem ter o mesmo nome. Os objetos SQL — tabelas, campos etc. — em termos dos quais as instruções SQL são compiladas não são declarados no programa PL/I, mas sim no dicionário do Sistema R. De fato, esses objetos não são conhecidos pelo programa PL/I em si, e não podem ser referenciados por outras instruções que não as da SQL embutida.

5. Após a execução de qualquer instrução SQL, é retornado ao programa um indicador numérico de estado, em uma variável do sistema chamada SYR_CODE. Um valor zero indica que a instrução foi executada com sucesso; um valor + 100 indica que não foi encontrado dado no banco de dados que satisfaça à solicitação; um valor negativo indica erro. Por isso o programador deve testar o valor de SYR_CODE após cada instrução SQL, tomando a ação adequada caso o valor não seja o esperado. Esta etapa não está mostrada na Fig. 8.1.

6. Finalmente, enfatizamos o fato de que *qualquer* instrução SQL pode ser usada dentro de um programa de aplicação — não somente as instruções DML, mas também as instruções DDL, se desejado. Na prática, como seria de se esperar, são as instruções DML as que mais freqüentemente aparecem, mas não é difícil encontrar-se também instruções DDL; por exemplo, CREATE TABLE pode ser usada para se criar uma tabela que guarde resultados intermediários. Ainda assim, o restante deste capítulo irá se concentrar somente nas instruções DML.

A maioria das instruções SQL DML pode ser embutida em PL/I e COBOL de uma forma bastante direta (isto é, com apenas pequenas alterações em sua sintaxe). Entretanto, a instrução SELECT requer tratamento especial. A execução de uma instrução SQL SELECT faz com que seja retornada uma *tabela* ao usuário – uma tabela que, em geral, contém múltiplos registros. No entanto, linguagens como o PL/I não estão suficientemente equipadas para manusear mais de um registro por vez. É por isso necessário fornecer alguma espécie de ponte entre os dois níveis funcionais, e a SQL embutida oferece essa ponte por meio de um novo tipo de objeto chamado *cursor*. Vamos deixar a discussão detalhada sobre cursores para a Seção 8.3, e estudar primeiramente (Seção 8.2) as instruções que não o utilizam.

8.2 OPERAÇÕES NÃO ENVOLVENDO CURSORES

As instruções DML que não necessitam de cursores são:

- SELECT de tabela com uma só linha
- UPDATE (exceto a forma “current” – veja Seção 8.3)
- INSERT
- DELETE (novamente com exceção da forma “current”)

Daremos exemplos de cada uma dessas instruções separadamente.

8.2.1 SELECT de tabela com uma só linha – Recupere o status e a cidade do fornecedor cujo número de fornecedor se encontra na variável GIVENS\$ do programa.

```
$SELECT STATUS, CITY  
      INTO $RANK, $CITY  
     FROM S  
    WHERE S# = $GIVENS#;
```

No exemplo, se houver exatamente um par (STATUS, CITY) que satisfaça a SELECT, esse par de valores será entregue às variáveis do programa RANK e CITY conforme solicitado, e o código SYR_CODE será ajustado para zero. Se nenhum par (STATUS, CITY) satisfizer a SELECT, SYR_CODE será ajustado para +100. Se mais de um par (STATUS, CITY) satisfizer a SELECT, o programa estará em erro; SYR_CODE será ajustado para um valor negativo (neste caso -810).

8.2.2 UPDATE – Aumente o status de todos os fornecedores de Londres do valor fornecido na variável RAISE do programa.

```
$UPDATE S  
      SET STATUS = STATUS + $RAISE  
     WHERE CITY = 'LONDON';
```

Se nenhum registro S satisfizer a cláusula WHERE, SYR_CODE será ajustado para +100.

8.2.3 INSERT – Insira uma nova peça (número da peça, nome e peso dados pelas variáveis do programa PNO, PNNAME, PWT respectivamente; cor e cidade desconhecidas) na tabela P.

```
$INSERT INTO P (P#, PNNAME, WEIGHT):  
          <$PNO, $PNNAME, $PWT>;
```

8.2.4 DELETE – Remova todos os embarques dos fornecedores cuja cidade está determinada na variável CITY do programa.

```
$DELETE SP
  WHERE S# IN
    (SELECT S#
      FROM S
     WHERE CITY = $CITY);
```

Novamente SYR_CODE será ajustado para +100 se nenhum registro satisfizer à cláusula WHERE.

8.3 OPERAÇÕES ENVOLVENDO CURSORES

Vamos agora considerar o caso de SELECT quando esta seleciona um conjunto completo de registros, e não somente um. Precisa-se de um mecanismo que possibilite o acesso aos registros no conjunto um por um; os *cursos* fornecem esse mecanismo. O processo está esboçado no exemplo da Fig. 8.2, que tem como objetivo recuperar informações sobre fornecedores (S#, SNAME, e STATUS) de todos os fornecedores da cidade determinada pela variável Y do programa.

```
$LET X BE          /* define cursor X */
  SELECT S#, SNAME, STATUS
    INTO $S#, $SNAME, $STATUS
   FROM S
  WHERE CITY = $Y;

$OPEN X;           /* ativa e X */
DO WHILE more-to-come;
  $FETCH X;        /* obtém novo fornecedor */
  . . .
END;
```

Fig. 8.2 Recuperação de múltiplos registros.

Vamos agora considerar as operações do cursor com maiores detalhes. Um cursor é *definido* por meio de uma instrução LET, que tem como forma geral

```
$LET cursor-name BE embedded-SELECT-statement;
```

Como um exemplo, veja a Fig. 8.2. A instrução LET é declarativa, não executável. A instrução LET na Fig. 8.2 declara X como um cursor, com uma instrução SELECT associada como especificado (note que SELECT pode incluir “\$-references”). Um programa pode conter qualquer número de instruções LET, tendo que ser cada uma para um cursor diferente.

São fornecidas três instruções para operarem sobre os cursos: OPEN, FETCH, e CLOSE.

1. A instrução

```
$OPEN X;
```

torna o cursor X *ativo*. Efetivamente, é executada a instrução SELECT associada a X (usando o valor corrente da variável Y do programa); é então identificado um conjunto de registros, sendo o cursor X associado ao conjunto. O cursor X também identifica uma *posição* naquele conjunto, ou seja, a posição imediatamente anterior ao primeiro registro do conjunto. (Conjuntos de registros associados a um cursor são sempre considerados como possuindo uma ordenação — seja uma ordenação definida por cláusula ORDER BY na instrução LET, ou ordenação determinada pelo sistema na falta daquela cláusula.)

2. A instrução

```
$FETCH X;
```

avança o cursor X e o posiciona “sobre” o próximo registro do conjunto associado, transferindo então campos daquele registro para as variáveis do programa, de acordo com as cláusulas SELECT e INTO da instrução LET de X. (Após o OPEN, X está posicionado imediatamente antes do primeiro registro, de forma que o primeiro FETCH irá posicioná-lo sobre o primeiro registro e deste extrair os campos.) Normalmente a instrução FETCH é executada dentro de um *loop* do programa, como mostrado na Fig. 8.2.

3. A instrução

```
$CLOSE X;
```

desativa o cursor X; isto é, X não fica mais associado com o conjunto de registros selecionados quando ele foi ativado. Entretanto ele permanece associado com a mesma instrução SELECT, e se for novamente ativado voltará a ficar associado ao mesmo conjunto (provavelmente não exatamente o mesmo conjunto de antes, especialmente se o valor da variável Y do programa tiver mudado nesse intervalo).

Se não houver “próximo” registro quando FETCH for executado, SYR_CODE será ajustado para +100, e o cursor automaticamente desativado. Isto explica por que não há instrução CLOSE na Fig. 8.2.

Note, incidentalmente, que uma mudança no valor da variável Y do programa enquanto o cursor X se encontra no estado de ativo não tem efeito sobre o conjunto de registros endereçáveis por X (o conjunto com o qual X está associado correntemente).

Duas instruções adicionais podem incluir referências a cursos. São as formas “current” de UPDATE e DELETE. Se um cursor, digamos X, estiver posicionado sobre um determinado registro do banco de dados, então é possível UPDATE ou DELETE o “current de X”, isto é, o registro sobre o qual X está posicionado. Por exemplo

```
$UPDATE S  
SET STATUS = STATUS + $RAISE  
WHERE CURRENT OF X;
```

UPDATE e DELETE estão sujeitas a certas restrições, das quais a mais importante é que o registro a ser atualizado ou removido — isto é, o registro como ele é visto pelo usuário — tem que ser essencialmente idêntico ao registro real do banco de dados (por exemplo, ele não pode ser uma união).¹ Além disso, a instrução LET do cursor considerado (a) não

¹ Pode, entretanto, omitir campos do registro básico. Esta questão de restrições de atualização será considerada em maiores detalhes no Capítulo 9.

pode incluir cláusula ORDER BY, e (b) no caso de UPDATE, tem que incluir uma cláusula da forma

FOR UPDATE OF field-name [, field-name] ...

identificando todos os campos que podem ser os objetivos de uma cláusula SET em uma instrução "UPDATE CURRENT" para aquele cursor.

Um Exemplo Abrangente

Apresentaremos um exemplo abrangente (Fig. 8.3) para ilustrar o uso de cursores com todos os detalhes. O programa aceita quatro valores de entrada: um número de peça (GIVENP#), um nome de cidade (GIVENCIT), um incremento de status (GIVENINC), e um nível de status (GIVENLVL). O programa esquadrinha todos os fornecedores da peça identificada por GIVENP#. Para cada fornecedor, se a sua cidade for GIVENCIT, o status é aumentado de GIVENINC; caso contrário, se o status for menor do que GIVENLVL, o fornecedor é removido, juntamente com todos os seus embárques. Em todos os casos a informação do fornecedor é listada em uma impressora, indicando como aquele fornecedor específico foi manuseado pelo programa.

```
SQLLEX: PROC OPTIONS (MAIN);

$DCL GIVENP# CHAR(6);
$DCL GIVENCIT CHAR(15);
$DCL GIVENINC FIXED BIN(15);
$DCL GIVENLVL FIXED BIN(15);
$DCL S# CHAR(5);
$DCL SNAME CHAR(20);
$DCL STATUS FIXED BIN(15);
$DCL CITY CHAR(15);
DCL DISP CHAR(7);
DCL MORE_SUPPLIERS BIT(1);

$SYR;
/* expande a estrutura SYR de informações de retorno ao Sistema R */

$LET Z BE SELECT * INTO $$#, $SNAME, $STATUS, $CITY
      FROM S WHERE S# IN
          (SELECT S# FROM SP WHERE P# = $GIVENP#)
      FOR UPDATE OF STATUS;
/*      ORDER BY não é permitida devido a UPDATE/DELETE      */

ON CONDITION (DBERROR)
BEGIN;
    PUT SKIP LIST (SYR_CODE);
    /* na prática, provavelmente imprimiria informações adicionais      */
    /* para depuração; por exemplo variáveis do                      */
    /* programa                                         */
    GO TO QUIT;
END;
```

Fig. 8.3 SQL embutida – um exemplo completo (primeira parte).

```

GET LIST (GIVENP#, GIVENCIT, GIVENINC, GIVENLVL);

$BEGIN TRANSACTION;

$OPEN Z;
IF SYR_CODE != 0 THEN SIGNAL CONDITION (DBERROR);
MORE_SUPPLIERS = '1'B;
DO WHILE (MORE_SUPPLIERS);
  $FETCH Z;
  SELECT;
/* NB: SELECT - WHEN - WHEN - ... - END is a PL/I */
/* "'case'" statement, not a SQL SELECT statement */
  WHEN (SYR_CODE = 100)
    /* Z automatically CLOSED */
    MORE_SUPPLIERS = '0'B;
  WHEN (SYR_CODE < 0)
    SIGNAL CONDITION (DBERROR);
  WHEN (SYR_CODE = 0)
    DO;
      DISP = '';
      IF CITY = GIVENCIT THEN
        DO;
          $UPDATE S
          SET STATUS = STATUS + $GIVENINC
          WHERE CURRENT OF Z;
          IF SYR_CODE != 0
            THEN SIGNAL CONDITION (DBERROR);
          DISP = 'UPDATED';
        END;
      ELSE
        IF STATUS < GIVENLVL THEN
          DO;
            $DELETE SP
            WHERE S# = $$#;
            IF SYR_CODE < 0
              THEN SIGNAL CONDITION (DBERROR);
            $DELETE S
            WHERE CURRENT OF Z;
            IF SYR_CODE != 0
              THEN SIGNAL CONDITION (DBERROR);
            DISP = 'DELETED';
          END;
        PUT SKIP LIST (S#, SNAME, STATUS, CITY, DISP);
      END; /* WHEN (SYR_CODE = 0) */
    END; /* SELECT */
  END; /* DO loop */

$END TRANSACTION;
QUIT;
END; /* SQLEX */

```

Fig. 8.3 SQL embutida — um exemplo completo (segunda parte).

Observe as instruções especiais \$BEGIN TRANSACTION e \$END TRANSACTION na Fig. 8.3. \$BEGIN TRANSACTION tem que ser a primeira instrução SQL executada em um programa; \$END TRANSACTION tem que ser a última, senão o programa termina de forma anormal. O termo "transaction" refere-se à unidade de trabalho que o programa executa entre essas duas instruções. Se \$END TRANSACTION não for executada, o Sistema R irá automaticamente cancelar todas as alterações feitas pelo programa no banco de dados desde \$BEGIN TRANSACTION.

Observe também a instrução "\$SYR;" (após as declarações PL/I). Esta instrução faz com que o pré-compilador gere uma estrutura PL/I chamada SYR, que é usada para receber informações de retorno após a execução de uma instrução SQL. A variável de código de retorno SYR_CODE é um campo dessa estrutura. Detalhes sobre os outros campos ultrapassam o escopo deste livro.

8.4 INSTRUÇÕES DINÂMICAS

Como mencionamos no Capítulo 5, a SQL embutida fornece alguns dispositivos que facilitam o preparo de programas de aplicações *on-line* – isto é, programas que se destinam a suportar acesso *on-line* ao banco de dados a partir de um terminal de usuário final. Vejamos o que esse programa tem que fazer. *Grosso modo*, as etapas que ele tem que cumprir são:

1. Aceitar um comando do terminal;
2. Analisar o comando;
3. Emitir as instruções SQL apropriadas;
4. Retornar uma mensagem e/ou resultados ao terminal.

Se o conjunto de comandos que o programa tem que aceitar for muito pequeno, como por exemplo um programa que manuseie reservas de passagens aéreas, então o conjunto de instruções SQL possíveis a serem emitidas poderá também ser pequeno, e pode ser "ligado por fios" ao programa. Neste caso, as etapas 2 e 3 acima consistirão do simples exame lógico do comando de entrada e posterior desvio para a parte do programa que emite instruções SQL predefinidas. Se, por outro lado, puder haver uma grande variedade de entradas, pode não ser prático predefinir instruções SQL "Ligadas por fios" para cada comando possível. Ao invés disso, será provavelmente mais conveniente *construir* as instruções SQL dinamicamente (e depois executar essas instruções construídas dinamicamente).² As "instruções dinâmicas" da SQL embutida são fornecidas para ajudar nesse processo.

As duas principais instruções dinâmicas são PREPARE e EXECUTE. O uso dessas duas instruções está ilustrado no seguinte exemplo.³

² Esta é a forma exata como o UFI trabalha. UFI é um programa de aplicação *on-line* que aceita uma grande variedade de entradas, ou seja, qualquer instrução SQL válida (ou inválida!). Usa as facilidades da instrução dinâmica para construir instruções SQL embutidas correspondentes às entradas e executá-las.

A SQL usa a palavra chave AS no lugar de FROM na instrução PREPARE. Nós usamos FROM para maior clareza.

```

DCL SQLSOURCE CHAR(256);
      .
SQLSOURCE = 'DELETE SP WHERE QTY < 100';
$PREPARE SQLOBJ FROM SQLSOURCE;
$EXECUTE SQLOBJ;

```

Explicaremos este exemplo a seguir. A instrução de designação atribui à variável de seqüência de caracteres SQLSOURCE a representação de seqüência de caracteres de uma SQL. (Na prática, o processo de criar esta representação de seqüência de caracteres seria provavelmente bem mais complicado.) A instrução PREPARE passa então esta seqüência ao pré-compilador RDS, que corre o seu processo normal de análise, otimização e geração de código, criando uma versão da instrução em linguagem de máquina, chamada SQLOBJ. (Este nome é arbitrário e é especificado na PREPARE.) Finalmente, a instrução EXECUTE faz com que esta rotina em linguagem de máquina seja executada causando (neste exemplo) a ocorrência real das remoções.

Observe, incidentalmente, que SQLSOURCE na instrução PREPARE é uma referência ordinária a uma variável, não uma “\$-reference”.

Uma vez processada pelo PREPARE, uma instrução SQL dinamicamente gerada pode ser processada pelo EXECUTE muitas vezes. A instrução gerada pode ser substituída por outra, emitindo-se novamente o PREPARE com o mesmo objetivo e uma fonte diferente.

O procedimento ilustrado é adequado para a geração dinâmica e execução de todas as instruções SQL “regulares”, *exceto* SELECT. (O termo “regulares” exclui as instruções de cursores, tais como OPEN e CLOSE, e as próprias instruções dinâmicas que descrevemos nesta seção). A razão por que SELECT é diferente vem do fato de ela retornar dados ao usuário; todas as outras instruções retornam apenas um indicador de estado (valor de SYR_CODE).

Um programa usando SELECT tem que saber algumas coisas sobre os dados que serão recuperados; no final, tem que especificar um conjunto de variáveis objetivo na cláusula INTO daquela SELECT. Em outras palavras, ele precisa saber pelo menos quantos campos serão recuperados e de que tipo de dados eles são. Se o SELECT for gerado dinamicamente em resposta a um comando do terminal, o programa em geral não pode ter esta informação antecipadamente. O que ele faz, então, é o seguinte:

1. Cria e submete ao PREPARE a instrução SELECT *sem* cláusula INTO.
2. Pergunta então ao Sistema R sobre que resultados podem ser esperados daquele SELECT, usando outra instrução “dinâmica” chamada DESCRIBE.
3. Depois, coloca disponível uma série de variáveis objetivo para receber os resultados, de acordo com o que lhe foi informado pela DESCRIBE.
4. Finalmente, usa OPEN, FETCH, e CLOSE para recuperar os registros do resultado um a um, fornecendo uma cláusula INTO na FETCH (pois não havia sido especificada na SELECT).

Vamos tornar o processo esboçado algo mais concreto, estruturando um procedimento PL/I para executar as etapas 1 a 4 em seqüência (Fig. 8.4). A SELECT específica mostrada e os nomes SQLSOURCE, SQLOBJ, D, e E são todos arbitrários. Observe que foram usadas OPEN e FETCH sobre a SELECT preparada, como se ela fosse um cursor.

```

DCL SQLSOURCE CHAR(100);

SQLSOURCE = 'SELECT * FROM SP WHERE QTY > 100';
$PREPARE SQLOBJ FROM SQLSOURCE;
$DESCRIBE SQLOBJ INTO D;

/* D é um indicador de localização do PL/I que designa uma estrutura      */
/* que será preenchida pelo DESCRIBE. Após o                                */
/* DESCRIBE, a estrutura conterá a seguinte informação                      */
/* (algo simplificada):                                                 */
/*   número de campos obtidos pelo SELECT;                                */
/*   para cada campo —                                         */
/*     tipo de dado                                         */
/*     comprimento                                         */
/* usando a informação retornada pelo DESCRIBE, o                         */
/* programa pode alocar uma variável objetivo para cada                   */
/* campo a ser retornado, usando (por exemplo) a capacidade normal       */
/* de alocação de armazenamento do PL/I. O programa                      */
/* constrói então outra estrutura, fornecendo as seguintes                 */
/* informações (algo simplificadas): */
/*   para cada variável objetivo */
/*     tipo de dado                                         */
/*     comprimento                                         */
/*     endereço                                           */
/* O programa prepara um indicador de localização, diagrama E,           */
/* para indicar essa estrutura.                                              */
/* */
```

```

$OPEN SQLOBJ;
DO WHILE more-to-come;
  $FETCH SQLOBJ INTO E;
  /* observe que E identifica uma descrição das variáveis                */
  /* INTO, não as variáveis em si.                                         */
  *****
END;
```

Fig. 8.4 Manuseio de SELECT dinâmica.

Instruções dinamicamente geradas não podem conter “\$-references”. No entanto, elas podem conter *parâmetros*, marcados na representação de seqüência de caracteres por pontos de interrogação. Os parâmetros podem aparecer onde as “\$-references” pudessem aparecer. Por exemplo,

```

SQLSOURCE = 'DELETE SP WHERE QTY > ? AND QTY < ?';
$PREPARE SQLOBJ FROM SQLSOURCE;
```

Os argumentos que substituem os parâmetros são especificados quando a instrução preparada dinamicamente é executada; por exemplo,

```
$EXECUTE SQLOBJ USING $LOW, $HIGH;
```

Os argumentos e parâmetros são causados por suas posições ordinais. Por isso, no exemplo, o DELETE que é executado é equivalente a

```
$DELETE SP WHERE QTY > $LOW AND QTY < $HIGH;
```

Como sempre, SELECT requer tratamento especial. Se a instrução preparada for um SELECT – por exemplo,

```
SQLSOURCE = 'SELECT * FROM SP WHERE QTY > ? AND QTY < ?';
$PREPARE SQLOBJ FROM SQLSOURCE;
```

então os argumentos são especificados no OPEN (lembre-se de que não é usado EXECUTE neste caso) – por exemplo

```
$OPEN SQLOBJ USING $LOW, $HIGH;
```

8.5 DISCUSSÃO

Concluiremos nossas discussões sobre a SQL embutida fornecendo uma série de observações sobre a abordagem do Sistema R à programação de aplicações para bancos de dados.

1. O uso de uma linguagem que é essencialmente a mesma tanto para o acesso *on-line* via UFI como para o acesso a partir de programas de aplicações tem uma série de vantagens. Por um lado, facilita grandemente a comunicação entre usuários do UFI e programadores de aplicações. Por outro, fornece uma ferramenta conveniente para desenvolvimento e depuração de programas; é fácil gerar-se bancos de dados simples para testes usando o UFI e acionar as porções do programa que lidam com bancos de dados – pelo menos as operações que não lidam com cursorse – a partir de um terminal. Além disso, a linguagem é essencialmente a mesma, não importando que a básica seja PL/I ou COBOL.

2. O fato de serem as solicitações do Sistema R ao banco de dados – isto é, as instruções SQL – compiladas (ou melhor, pré-compiladas) significa que o Sistema R pode ter uma vantagem de desempenho em tempo de execução sobre outros sistemas que adotam uma abordagem interpretativa mais convencional [8.1, 8.2]. As operações de análise da solicitação, amarração dos nomes do nível fonte a identificadores internos do sistema, seleção dos caminhos de acesso (otimização), e verificação de autorização podem ser todas executadas ao tempo de pré-compilação, sendo portanto removidas do caminho em tempo de execução. (O caminho em tempo de execução é a seqüência de instruções necessária para executar as operações sobre o banco de dados em tempo de execução.) Uma vez que o caminho em tempo de execução tem que ser percorrido a cada solicitação, e tais solicitações freqüentemente ocorrem dentro de *loops* de programas, a economia pode ser considerável.

3. Mesmo no caso de instruções construídas dinamicamente, onde tudo necessariamente se encontra no caminho em tempo de execução, a abordagem de compilação pode render frutos. Isto porque o código gerado é talhado para a solicitação específica, ao invés de ser generalizado (e portanto parcialmente interpretativo). De fato, está declarado que a compilação é mais rápida do que a interpretação assim que se torna necessário o acesso a mais do que uma quantidade muito pequena de registros armazenados no banco de dados [8.1].

4. Uma desvantagem da SQL como uma linguagem de programação de aplicações é que ela está muito frouxamente acoplada à básica. De fato, ela é naturalmente uma linguagem totalmente diferente. Os objetos SQL (por exemplo, tabelas, campos, cursorse) não são conhecidos e não podem ser referenciados no ambiente básico; e os objetos da

básica só podem ser referenciados no ambiente SQL de uma maneira específica (“\$-references”). As regras sobre coisas tais como formação de nomes, resolução de nomes, qualificação de nomes, avaliação de expressões e outras mudam todas ligeiramente quando o programador cruza a fronteira entre a SQL e a linguagem básica. Talvez mais significativo seja o fato de a SQL não tirar vantagem das construções já existentes na linguagem básica para coisas tais como controle de *loops*, estruturas de dados, manuseio de exceções, e passagem de argumentos. Como consequência, os programas SQL tendem a ser bastante não-estruturados e não muito concisos.

EXERCÍCIOS

8.1 Usando o banco de dados de fornecedores-peças-projetos, escreva um programa com instruções SQL embutidas para listar todos os registros de fornecedores na ordem de número de fornecedor. Cada registro de fornecedor deverá estar imediatamente seguido na listagem por todos os registros de peças que são supridas por aquele fornecedor na ordem de número de peça.

8.2 Para que você acha que é usada a cláusula FOR UPDATE?

8.3 Dada a tabela

COMPONENT (MAJORP#, MINORP#, QTY)

(compare com a Fig. 4.4), escreva um programa SQL para listar todas as peças componentes de uma determinada peça em todos os níveis (o problema de “explosão da peça”).

8.4 O que você acha que aconteceria se uma instrução FETCH ou uma SELECT de uma só linha de tabela tentasse recuperar um valor nulo?

REFERÊNCIAS E BIBLIOGRAFIA

8.1 R. A. Lorie and B. W. Wade. “The Compilation of a Very High Level Data Language.” IBM Research Report RJ2008 (maio de 1977).

Descreve a abordagem de pré-compilação.

8.2 D. D. Chamberlin et al. “Support for Repetitive Transactions and Ad-Hoc Query in System R.” IBM Research Report RJ2551 (maio de 1979).

Fornece mais detalhes sobre o pré-compilador e algumas informações sobre desempenho.

8.3 R. A. Lorie and J. F. Nilsson. “An Access Specification Language for a Relational Data Base System.” IBM J. R & D. 23, nº 3 (maio de 1979).

Para uma dada instrução SQL, o componente otimizador da RDS gera um programa em uma linguagem interna chamada ASL (Access Specification Language). Esta linguagem serve como interface entre o otimizador e o gerador de código, que converte um programa em ASL para código de máquina (incluindo as chamadas da RSS). A ASL consiste de operadores como “scan” e “insert” sobre objetos tais como índices e arquivos armazenados. A ASL torna o processo total de tradução mais administrável, pois divide-o em subprocessos bem definidos.

8.4 P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. “Access Path Selection in a Relational Database System.” Proc. 1979 ACM SIGMOD International Conference on Management of Data (maio de 1979).

Fornece detalhes do otimizador.

9

O Nível Externo do Sistema R

9.1 INTRODUÇÃO

No nível externo da arquitetura ANSI/SPARC, o banco de dados é percebido como uma “visão externa”, definida por um esquema externo. Usuários diferentes podem ter visões diferentes. Como explicamos no Capítulo 5, entretanto, o termo *visão* está reservado no Sistema R para significar uma *tabela derivada* (e não uma tabela básica); portanto, a versão do Sistema R da “visão externa” ANSI/SPARC é, tipicamente, uma coleção de tabelas, algumas delas tabelas básicas e algumas visões. O “esquema externo” consiste de definições daquelas tabelas básicas e visões. Neste capítulo, vamos examinar com alguns detalhes a construção da visão do Sistema R.

A estrutura ANSI/SPARC é muito genérica, permitindo muitas variantes entre os níveis externo e conceitual. Em princípio, mesmo os *tipos* de estruturas suportadas nos dois níveis podem ser diferentes; por exemplo, o nível conceitual pode estar baseado em relações *n*-árias, enquanto que um determinado usuário pode ter uma visão externa desse banco de dados como hierárquico. Na prática, entretanto, a maioria dos sistemas implementados usa o mesmo tipo de tipo de estrutura como base para os dois níveis, e o Sistema R não constitui uma exceção — uma visão é ainda uma tabela, como uma tabela básica. E, como o mesmo tipo de objeto é suportado em ambos os níveis, a mesma linguagem de manipulação (isto é, a SQL DML) aplica-se também aos dois níveis.

9.2 VISÕES

Uma visão pode ser vista como uma tabela *virtual*, isto é, uma tabela que realmente não existe como tal, mas sim como uma derivação de uma ou mais tabelas básicas. Em outras palavras, não há arquivo armazenado que represente diretamente a visão em si. Ao invés, uma *definição* da visão fica armazenada no dicionário. A definição da visão mostra como ela é derivada das tabelas básicas.

Em princípio, qualquer tabela derivável pode ser definida como uma visão. O processo de derivação pode envolver a projeção de certos campos de uma tabela básica, ou a união de duas tabelas básicas, ou ainda a execução de qualquer seqüência de projeções,

uniões e operações similares sobre qualquer coleção de tabelas básicas. Uma visão é criada pela execução da instrução DEFINE VIEW, que tem como forma geral¹

```
DEFINE VIEW view-name
    [ ( field-name [, field-name ] ... ) ]
        AS SELECT-statement
```

Por exemplo,

```
DEFINE VIEW LONDON_SUPPLIERS
    AS SELECT S#, SNAME, STATUS
        FROM S
        WHERE CITY = 'LONDON'
```

Este exemplo define uma visão que é uma projeção de um subconjunto horizontal da tabela básica S. A visão chama-se LONDON_SUPPLIERS, e tem três campos, de nomes S#, SNAME, e STATUS, herdados dos nomes da tabela S. Se quiséssemos, poderíamos ter dado novos nomes aos campos, especificando, digamos "(LS#, LSNAME, LSTAT)", após o nome da visão LONDON_SUPPLIERS na DEFINE VIEW. Os nomes de campos têm que ser especificados caso os nomes herdados não sejam únicos – por exemplo, se a visão for uma união de duas tabelas, e aquelas tabelas possuírem alguns nomes comuns de campos – ou se algum “campo” selecionado for o resultado de uma expressão aritmética ou de uma função integrada. Daremos um exemplo de cada um desses casos.

```
DEFINE VIEW CITY_PAIRS ( SCITY, PCITY )
    AS SELECT UNIQUE S.CITY, P.CITY
        FROM S, SP, P
        WHERE S.S# = SP.S#
            AND SP.P# = P.P#
    
```



```
DEFINE VIEW PQ ( P#, SUMQTY )
    AS SELECT P#, SUM(QTY)
        FROM SP
        GROUP BY P#
```

A instrução SELECT em uma definição de visão não pode incluir ORDER BY, nem conter referências a variáveis de programas (“\$-references”).

Definida uma visão, o usuário pode utilizá-la como se ela fosse uma tabela real, sujeita a certas restrições (a serem discutidas). Por exemplo,

```
SELECT *
    FROM LONDON_SUPPLIERS
    WHERE STATUS < 50
    ORDER BY S#
```

¹

Em termos ANSI/SPARC, DEFINE VIEW combina a função do esquema externo (descrevendo o objeto externo) com a função de definição do mapeamento (especificando o mapeamento ao nível conceitual).

A execução bem-sucedida de uma instrução DEFINE VIEW faz com que a visão seja armazenada no dicionário do Sistema R. A instrução SELECT dentro daquela definição *não* é executada nesse momento. Ao invés disso, o que ocorre é que quando o usuário executa um SELECT (ou UPDATE ou DELETE) sobre a visão, a operação e a SELECT da visão são *combinadas* para formar uma instrução modificada que opera sobre os dados básicos.² Por exemplo, a instrução SELECT acima será combinada com a definição de LONDON_SUPPLIERS para produzir

```
SELECT S#, SNAME, STATUS  
FROM S  
WHERE STATUS < 50  
AND CITY = 'LONDON'  
ORDER BY S#
```

A instrução modificada é então processada normalmente.

Uma vez que as definições de visões são expressas por meio de instruções SELECT, e como as instruções SELECT podem selecionar dados tanto de visões como de tabelas básicas, torna-se possível definir uma visão em termos de outras. Por exemplo,

```
DEFINE VIEW LONDON_NAMES  
AS SELECT SNAME  
FROM LONDON_SUPPLIERS
```

Também é possível destruir uma visão existente a qualquer momento:

```
DROP VIEW view-name
```

A definição da visão será removida do dicionário. As tabelas básicas envolvidas não serão afetadas em nada; mas se outras visões estiverem definidas em função desta, elas serão também automaticamente destruídas. Da mesma forma, se uma tabela básica for destruída, todas as visões definidas com base nela serão também automaticamente destruídas.

Concluindo esta seção, observemos que, no Sistema R, os campos de uma visão herdam o tipo de dado da tabela básica envolvida. O campo STATUS de LONDON_SUPPLIERS, por exemplo, tem o tipo de dado SMALLINT, como na tabela S. Um mecanismo mais geral permitiria que campos das visões tivessem tipos de dados diferentes daqueles da tabela básica.

9.3 OPERAÇÕES DML SOBRE VISÕES

Uma visão é uma “janela” ao dado real, não uma cópia separada daquele dado. As mudanças do dado real são visíveis através da visão; e, como nós explicamos na seção anterior, as operações sobre a visão são convertidas em operações sobre os dados reais. Para operações SELECT, esta conversão é sempre possível, isto é, do ponto de vista de uma recuperação, é certamente verdadeiro que uma visão pode ser qualquer tabela derivável. Mas a situação torna-se diferente nas operações de atualização, como veremos agora.

No Sistema R, uma visão que possa aceitar atualizações tem que ser derivada de uma única tabela básica. Além disso, ela tem que satisfazer às seguintes restrições:

- C1. Cada linha distinta da visão tem que corresponder a uma distinta e univocamente identificável linha da tabela básica.

² O processo de combinação é efetuado pelo pré-compilador RDS antes da etapa de otimização.

C2. Cada coluna distinta da visão tem que corresponder a uma distinta e univocamente identificável coluna da tabela básica.

Em outras palavras, as únicas diferenças permitidas entre a visão e a tabela básica são que, na visão: (a) linhas individuais (as que não satisfaçam ao predicado definidor) podem ser omitidas, e/ou (b) colunas individuais podem ser omitidas. A visão *não* pode ser uma projeção genuína (com as duplicações eliminadas), nem uma união, nem uma junção; não pode também envolver GROUP BY, nem conter campos calculados.³

Deve ficar claro que se uma visão satisfizer às restrições C1 e C2, então qualquer atualização sobre ela poderá ser facilmente mapeada com uma atualização sobre a tabela básica correspondente; em outras palavras, essas visões são “atualizáveis”. Para mostrar o oposto — que visões violando C1 e C2 não são atualizáveis e que consequentemente as restrições são razoáveis — consideremos alguns exemplos.

Exemplo 1

```
DEFINE VIEW V1 AS  
    SELECT UNIQUE COLOR, CITY  
    FROM P
```

V1 é uma projeção da tabela P, com as duplicações eliminadas. Se a tabela P for como a mostrada na Fig. 4.7, o valor (extensão) de V1 — isto é, o subconjunto de P visível via V1 — será como mostrado na Fig. 9.1. Ao lado de cada linha desta extensão, estamos mostrando os valores P# correspondentes.

V1	COLOR	CITY	
	Red	London	P1, P4, P6
	Green	Paris	P2
	Blue	Rome	P3
	Blue	Paris	P5

Fig. 9.1 Extensão da visão V1.

Deve ficar claro que V1 não pode suportar operações de INSERT. (Inserções à tabela base P requerem que o usuário especifique um valor P#.) Teoricamente poderiam ser definidas operações de DELETE e UPDATE (para remover ou atualizar *todas* as linhas correspondentes em P), mas essas operações poderiam perfeitamente ser expressas diretamente em termos de P, ou pelo menos uma visão de P que inclua P#, e provavelmente um usuário que esteja atualizando ou removendo peças estará interessado em saber exatamente que peças estão sendo afetadas.

3

Uma exceção a essas restrições é que são permitidas operações DELETE sobre visões que incluem campos calculados (violando C2, mas não C1), sendo as operações UPDATE permitidas sobre os outros (não calculados) campos daquela visão.

Exemplo 2

```
DEFINE VIEW V2 AS
    SELECT P#, CITY
    FROM SP, S
    WHERE SP.S# = S.S#
```

V2 envolve uma união das tabelas SP e S em S#. A extensão correspondente à Fig. 4.7 está mostrada na Fig. 9.2; os valores de S# correspondentes a cada linha aparecem ao lado.

V2 não pode suportar qualquer operação de atualização. A inserção de um novo par (P#, CITY) envolveria a inserção de uma nova linha SP, para a qual é exigido um valor S#. A remoção de um par (P#, CITY) iria requerer a remoção de alguma linha SP, mas nem sempre fica muito claro qual; por exemplo, o par (P2, Paris) poderia corresponder tanto a S2 como a S3. A atualização de um par (P#, CITY) – digamos, mudança de (P2, Paris) para (P2, London) – poderia significar tanto que o fornecedor envolvido (qual?) mudou de localização, como que a peça agora está sendo fornecida por um fornecedor diferente (novamente, qual?). A situação seria ainda pior se a SELECT na DEFINE VIEW tivesse especificado UNIQUE.

V2		P#	CITY	S1	P1	Paris	S2
P1	London	S1	P2	Paris	S2		
P2	London	S1	P2	Paris	S3		
P3	London	S1	P2	London	S4		
P4	London	S1	P4	London	S4		
P5	London	S1	P5	London	S4		
P6	London	S1					

Fig. 9.2 Extensão da visão V2

Exemplo 3

```
DEFINE VIEW V3 ( P#, SUMQTY ) AS
    SELECT P#, SUM(QTY)
    FROM SP
    GROUP BY P#
```

V3 envolve GROUP BY e uma função integrada. Sem que se tenha olhado qualquer extensão, fica óbvio que INSERT não pode ser manipulado; nem UPDATE, se o campo a ser modificado for SUMQTY. Poderiam ser definidas operações DELETE e UPDATE no campo P# (para remover ou atualizar *todas* as linhas correspondentes de SP), mas os comentários feitos sob o Exemplo 1 a respeito deste tópico valem novamente aqui.

Exemplo 4

```
DEFINE VIEW REDPARTS AS
    SELECT P#, PNAME, WEIGHT, COLOR
    FROM   P
    WHERE  COLOR = 'RED'
```

REDPARTS satisfaz às restrições e é portanto atualizável.⁴ Entretanto, servirá para ilustrar três pontos adicionais. Antes, mostremos (na Fig. 9.3) a extensão de REDPARTS correspondente à tabulação de P da Fig. 4.7.

REDPARTS	P#	PNAME	WEIGHT	COLOR
	P1	Nut	12	Red
	P4	Screw	14	Red
	P6	Cog	19	Red

Fig. 9.3 Extensão da visão REDPARTS.

O primeiro ponto é que um INSERT bem-sucedido sobre REDPARTS terá que gerar um valor nulo para o campo ausente CITY (naturalmente, este campo não pode ter sido definido com a opção NONULL). Segundo, vê-se da Fig. 9.3 que o usuário poderia criar uma linha via visão REDPARTS que tenha um valor P# de, digamos, P3, que não seja uma duplicata do ponto de vista da visão, mas que seja uma duplicata no que tange à tabela básica envolvida. Claramente esta operação não pode ser autorizada; tem que ser rejeitada, tal como se tivesse sido aplicada diretamente à tabela P.

Por fim, observe o seguinte UPDATE

```
UPDATE REDPARTS
SET    COLOR = 'BLUE'
WHERE  P# = 'P1'
```

Seria este UPDATE aceito? Se for, terá o efeito de remover a peça P1 da visão, pois ela não mais satisfaz ao predicado definidor. (Por outro lado, a verificação de cada atualização feita por meio de uma visão contra tal violação tornar-se-ia muito custosa.) No Sistema R, essas atualizações são aceitas e o registro não desaparece. Da mesma forma, um

4

Esta afirmativa ainda seria verdadeira mesmo que P# fosse excluído da visão, pois o Sistema R não elimina linhas duplicadas a menos que seja especificado UNIQUE. Mas não está claro se os usuários teriam possibilidade de atualizar essa visão, já que não há garantia de que uma determinada linha da visão corresponda a qualquer peça específica – isto é, o usuário não tem como especificar exatamente que peça deva ser atualizada. (Veja os comentários da visão V1 [Exemplo 1]. V1 também seria atualizável no Sistema R se “UNIQUE” fosse omitida da definição de V1.) Seria preferível substituir a restrição C1 por uma restrição de que a visão tenha que incluir a chave primária da tabela básica. Infelizmente, como chamamos a atenção no Capítulo 6, o Sistema R não toma conhecimento das chaves.

INSERT sobre uma visão pode ser aceito mesmo que o novo registro não satisfaça ao predicado da visão, mas o registro inserido não será visível por meio daquela visão.

Está claro que a filosofia do Sistema R sobre atualizações de visões é bastante própria; há situações nas quais o Sistema R proíbe uma atualização que, teoricamente, poderia ser suportada. A dificuldade reside em estabelecer precisamente que situações são essas. Diversos pesquisadores têm tentado definir um conjunto de regras mais sistemático e preciso [9.4, 9.5, 9.6]. Outros sugeriram que o sistema não devesse ter regras fixas, mas que o DBA (ou quem seja responsável pela definição das visões) definisse o efeito de cada tipo de atualização sobre cada visão [9.7, 9.8]. Cada uma dessas abordagens tem suas consequências. A abordagem do Sistema R de ter uma única regra fixa e simples provavelmente ganha por sua simplicidade mais do que perde devido à sua ligeira perda de função — embora, como veremos na próxima seção, haja certas situações em que a regra pode ser algo relaxada com vantagem.

9.4 VISÕES E INDEPENDÊNCIA DE DADOS

Nesta seção iremos considerar a contribuição prestada pelas visões à provisão da independência de dados — não só especificamente no Sistema R, mas também nos sistemas de bancos de dados em geral, embora continuemos a usar o Sistema R como um exemplo sempre que apropriado. Como mencionamos no Capítulo 1, há muitos níveis de independência de dados. A separação entre as camadas conceitual e interna do sistema fornece um nível maior, ou seja, independência dos detalhes de armazenamento (algumas vezes chamado de independência dos dados físicos). De forma semelhante, a separação entre as camadas externa e conceitual — o mecanismo da visão, no Sistema R — fornece outro nível maior, ou seja, independência das mudanças do esquema conceitual (algumas vezes chamada de independência dos dados lógicos [9.11]). Verifiquemos duas grandes classes de mudanças que podem ocorrer no esquema conceitual: crescimento e reestruturação.

Crescimento

À medida que cresce a abrangência do banco de dados — isto é, à medida que novos tipos de informações lhe são adicionados —, o esquema conceitual cresce também. Há dois tipos possíveis de crescimento que podem ocorrer no esquema conceitual:

1. Expansão de uma tabela básica existente para incluir um novo campo (correspondente à adição de nova informação relativa a algum tipo de entidade existente); e
2. Inclusão de uma nova tabela básica (correspondente à adição de um novo tipo de entidade).

Entretanto, nenhuma dessas mudanças deve afetar as visões existentes; por definição, aquelas visões não conterão referências aos novos campos ou tabelas.⁵ Assim, o mecanismo da visão pode isolar os usuários dos efeitos do crescimento do banco de dados. Chamamos a esse isolamento — possivelmente o aspecto que considerado isoladamente é o mais importante da independência de dados — de *imunidade ao crescimento*.

⁵ Na realidade, as instruções SQL são impermeáveis a essas mudanças, mesmo quando atuando sobre tabelas básicas ao invés de visões, *exceto* no caso de "SELECT *" (cujo significado pode se modificar se a tabela envolvida receber um novo campo).

Reestruturação

Ocasionalmente, torna-se necessário reestruturar o esquema conceitual de forma que, embora o conteúdo global de informações se mantenha o mesmo, se modifique a colocação dessas informações dentro do esquema — isto é, sejam alteradas as alocações de campos nas tabelas. Essa alteração é geralmente problemática, mas algumas vezes desejável. Uma classe importante de reestruturação, e a única que discutiremos (mas veja o Exercício 9.4), é a substituição de uma determinada tabela por duas de suas projeções, de forma que a tabela original possa ser recomposta unindo-se essas projeções. Por exemplo, a tabela S de fornecedores poderia ser substituída pelas duas tabelas seguintes:

```
SX (S#, SNAME, CITY)
SY (S#, STATUS)
```

(A tabela original S é a união natural de SX e SY em S#. Uma razão possível para a mudança poderia ser objetivando simplificar o controle de autorização.) Que efeito causará esta alteração sobre os usuários existentes? Do ponto de vista de recuperação, o efeito será *nenhum*, se definimos a antiga tabela S como a visão que se segue:

```
DEFINE VIEW S (S#, SNAME, STATUS, CITY) AS
    SELECT SX.S#, SNAME, STATUS, CITY
    FROM   SX, SY
    WHERE  SX.S# = SY.S#
```

Desta forma, qualquer operação SELECT existente sobre S continuará a funcionar tal como antes (embora requeira análise adicional durante a pré-compilação e maior sobrecarga de tempo durante a execução). No entanto, como explicado na Seção 9.3, as operações de atualização sobre S não funcionarão mais (em outras palavras, um usuário que efetue atualizações não está imune a este tipo de mudança) — embora, neste caso, as restrições sobre as atualizações da visão pudessem ser relaxadas, pois há uma correspondência de um-para-um entre a tabela SX e a tabela SY, e o efeito de todas as possíveis atualizações sobre S está claramente definido em termos de SX e SY. (Você concorda com esta afirmativa?)

Como outro exemplo, suponhamos que o banco de dados originalmente incluisse a tabela

```
SPT (S#, P#, QTY, STATUS)
```

(provavelmente um projeto não muito bom, como veremos no Capítulo 14, mas o projetista tem que ter o direito de errar de vez em quando), e suponhamos que SPT seja subsequentemente substituída por suas duas projeções

```
SP (S#, P#, QTY)
ST (S#, STATUS)
```

Novamente a tabela original SPT pode ser definida como uma visão, obtida da união das duas novas tabelas SP e ST em S#, e novamente as operações de recuperação ainda irão funcionar. Desta vez, no entanto, as duas projeções não estão em uma correspondência de um-para-um entre si, não havendo por isso possibilidade de se manusear automaticamente atualizações arbitrárias em termos de SPT.

9.5 RESUMO

Concluiremos este capítulo sumarizando em termos gerais vantagens de se separar a visão de dados de um usuário (definida por um esquema externo) da visão da comunidade (definida pelo esquema conceitual).

A maioria desses itens aplica-se a qualquer sistema que inclua esta separação, e não apenas ao Sistema R.

- Os usuários ficam imunes ao crescimento do banco de dados.
- Os usuários podem ficar imunes à reestruturação do banco de dados.
- A percepção do usuário fica simplificada.

É óbvio que o esquema externo permite que o usuário focalize somente os dados de seu interesse. O que talvez não seja tão óbvio, é que, pelo menos para recuperação, o esquema externo pode também simplificar consideravelmente as operações DML do usuário. Particularmente a necessidade de operações DML para saltar de tabela em tabela fica bastante reduzida, porque o usuário pode ter uma visão na qual todas as tabelas envolvidas estejam unidas. Como exemplo, consideremos a visão V2 da Seção 9.3, e compare a SELECT necessária para se encontrar números de peças disponíveis em Londres usando aquela visão com a SELECT necessária para se obter o mesmo resultado a partir das tabelas básicas envolvidas. Com efeito, o processo complexo de seleção é transferido da DML para a DDL (embora a distinção entre as duas seja pouco clara em uma linguagem como a SQL).

- O mesmo dado pode ser visto por usuários diferentes de maneiras diferentes.

Esta consideração é importante quando existem muitas categorias diferentes de usuários, todos interagindo sobre um único banco de dados integrado.

- Segurança automática fornecida aos dados secretos

“Dados secretos” são tabelas e campos não incluídos no esquema externo. Esses dados devem ficar claramente garantidos contra qualquer acesso dos usuários daquele esquema específico. Por isso, a obrigatoriedade de que os usuários tenham acesso ao banco de dados via um esquema externo é um meio simples, porém efetivo, de autorização de controle.

EXERCÍCIOS

9.1 Defina a relação SP (Fig. 4.8) como uma visão, do Sistema R, da relação SPJ (veja o Exercício 6.2).

9.2 Projete um mecanismo para definir visões externas *hierárquicas*, dado um esquema conceitual relacional.

9.3 Para que possam ser atualizadas, as relações no nível externo geralmente têm que ser essencialmente as mesmas que a relação correspondente no nível conceitual. Interprete esta restrição para o caso de uma visão externa hierárquica (definida, novamente, em termos de um esquema conceitual relacional).

9.4 Suponha que um esquema conceitual está reestruturado de tal forma que as relações A e B foram substituídas pela sua união natural C. Até que ponto pode o mecanismo da visão dissimular esta reestruturação dos usuários existentes?

REFERÊNCIAS E BIBLIOGRAFIA

9.1 C. J. Date and P. Hopewell. “File Definition and Logical Data Independence.” *Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control*.

Este artigo apresenta várias idéias semelhantes às deste capítulo, de uma forma um pouco diferente.

- 9.2 E. F., Codd. "Recent Investigations into Relational Data Base Systems." *Proc. IFIP Congress 1974*. Também em *Proc. ACM Pacific Conference, San Francisco* (abril de 1975), disponível na ACM Golden Gate Chapter, P. O. Box 24055, Oakland, California 94623.

Inclui uma discussão preliminar da teoria de suporte a múltiplas visões. Os outros itens discutidos são "Boyce/Codd Normal Form" (veja o capítulo 14), sublinguagens de dados, intercâmbio de dados e investigações necessárias. As "investigações necessárias" mais urgentes são ditas ser (parafraseando):

1. Desenvolvimento de técnicas de controle de concorrência;
2. Averiguação do desempenho alcançável em sistemas relacionais realmente grandes, com acesso e atualização concorrentes;
3. Desenvolvimento de uma teoria de visões;
4. Desenvolvimento de teoria de armazenamento, acesso e atualização; e
5. Desenvolvimento da viabilidade de subsistemas como o RENDEZVOUS [7.3-7.5].

- 9.3 D. D. Chamberlin, J. N. Gray, and I. L. Traiger. "Views, Authorization, and Locking in a Relational Data Base System." *Proc. NCC 44* (maio de 1975).

Inclui uma breve base lógica para a abordagem geral de atualização de visões no Sistema R.

- 9.4 S. J. P. Todd. "Automatic Constraint Maintenance and Updating Defined Relations". *Proc. IFIP Congress 1977*.

- 9.5 U. Dayal and P. A. Bernstein. "On the Updatability of Relational Views." Aiken Computation Laboratory, Harvard University, Cambridge, Mass. 02136 (1978).

- 9.6 A. L. Furtado and K. C. Sevcik. "Permitting Updates Through Views of Data Bases." Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, Brasil (1977).

- 9.7 K. C. Sevcik and A. L. Furtado. "Complete and Compatible Sets of Update Operations." Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, Brasil (1977).

- 9.8 A. L. Furtado. "A View Construct for the Specification of External Schemas." Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, Brasil (1978).

10

O Nível Interno do Sistema R

10.1 O RESEARCH STORAGE SYSTEM

O Research Storage System (RSS) é o método de acesso ou subsistema de armazenamento do Sistema R. Ele suporta a forma armazenada de uma tabela básica SQL, que nós chamamos de arquivo armazenado (para o RSS, simplesmente relação ou tabela). É sempre possível o acesso a um arquivo armazenado na seqüência do “sistema” (definida pela RSS); além disso, pode ser definida qualquer quantidade de índices sobre os arquivos armazenados, fornecendo assim caminhos adicionais de acesso a esses arquivos.¹ Os objetos RSS (arquivos armazenados, índices etc.) e os operadores associados constituem o Research Storage Interface, ou RSI, que é o interface usado como objetivo pelo RDS ao pré-compilar as solicitações SQL. Neste capítulo, vamos observar mais de perto o RSI, para dar uma idéia do que acontece “sob as cobertas” de um sistema em funcionamento. Nossa discussão, no entanto, será muito simplificada, e intencionalmente omitiremos muitos detalhes menores.

Além das funções aqui esboçadas, o RSS também fornece suporte ao controle de concorrência, administração de transações e checkpoint/restart do sistema. As referências [5.1, 5.3] fornecem detalhes.

10.2 SEGMENTOS E PÁGINAS

O banco de dados armazenado é dividido logicamente em um conjunto de *segmentos* separados. Cada segmento consiste de um número inteiro de *páginas* de 4096 bytes (o número de páginas de um determinado segmento varia dinamicamente, como nós veremos). A página é a unidade de transferência entre o banco de dados e o armazenamento primário.

¹ O RSS também fornece outro tipo de caminho de acesso, o *link* (elo de ligação), mas estes não são correntemente usados pelo RDS. Os links no RSS são basicamente cadeias de indicadores de localização, como as descritas no Capítulo 2; isto é, um link é um mecanismo para encadear um registro de um tipo com um conjunto de registros relacionados de outro tipo.

Quando é solicitada uma página ao banco de dados, o RSS a coloca em um espaço dentro do *buffer* da memória principal que é compartilhado por todos os usuários correntes, e informa o endereço daquele espaço ao componente que solicitou a página. O uso de página de tamanho único como unidade para todas as entradas e saídas simplifica a administração do *buffer* e fornece um interface adequado para a independência de dispositivos [10.1].

Os segmentos fornecem uma base para controle da alocação dos objetos de dados no armazenamento. Como explicamos no Capítulo 6, qualquer tabela básica (arquivo armazenado), juntamente com todos os índices referentes àquela tabela, tem que estar totalmente contida em um único segmento. Uma página qualquer pode conter tantos dados da tabela básica quanto dados de índices, mas não ambos. Mais significativamente, uma determinada tabela básica (com todos os seus índices) pode ser armazenada em um segmento do *tipo* apropriado. O Sistema R oferece três tipos diferentes de segmentos, com propriedades que são adequadas aos dados públicos, privativos e temporários, respectivamente. Basicamente, os dados do segmento *público* podem ser recuperados e compartilhados; os dados em segmentos *privativos* podem ser recuperados mas não compartilhados; e os dados em segmentos *temporários* não podem ser nem recuperados nem compartilhados. (“Recuperável” aqui está significando que os dados não serão perdidos no caso de uma falha; “compartilhado” está significando que os dados estão disponíveis a múltiplos usuários concorrentes.) Assim, por exemplo, a sobrecarga associada ao suporte completo para compartilhamento, necessário aos dados públicos, pode ser evitada para os dados temporários e privativos (que não requerem esses suportes). Observe, entretanto, que o tipo de um segmento é fixado durante a instalação do sistema, e não pode ser modificado.

Cada segmento possui um tamanho máximo predeterminado (geralmente muito grande), mas sempre ocupa somente o espaço físico de armazenamento realmente necessário aos objetos de dados que contém no momento. As páginas são alocadas aos segmentos à medida que se tornam necessárias (por exemplo, quando uma tabela cresce muito), e voltam a ser liberadas quando o segmento novamente se retrai (por exemplo, quando um índice é removido). É mantido um *mapa de páginas* para cada segmento, dando a localização física na memória secundária de cada página correntemente designada ao segmento.

No RSI, os segmentos são identificados por um *identificador de segmento* numérico, ou SID. As páginas são identificadas por um número de página dentro do segmento. Os nomes simbólicos de segmentos usados em CREATE TABLE e algumas outras operações SQL são transformados em SIDs pelo RDS, usando o dicionário do Sistema R. As páginas nunca são diretamente referenciadas na SQL.

10.3 ARQUIVOS E REGISTROS

Cada tabela básica é representada como um arquivo armazenado. Tal como segmentos, um arquivo armazenado é identificado no RSI por um identificador numérico chamado RID (R de relação). O RDS é responsável pela transformação dos nomes de tabelas SQL em RIDs.

Os registros do arquivo armazenado representam linhas da tabela. Cada registro é armazenado como uma seqüência de *bytes*. A seqüência de *bytes* consiste de um prefixo (contendo informações de controle, tal como RID do arquivo que a contém), seguido pela representação armazenada de cada campo no registro. No caso de campo de comprimento variável, a representação armazenada inclui uma indicação do comprimento do

campo. (Observe que todos os campos são considerados como simples sequências de bytes pelo RSS; qualquer interpretação dessas sequências como, por exemplo, ser um número de ponto flutuante, ocorre acima do RSI pelo RDS.)

Da mesma forma que segmentos e arquivos, os registros individuais possuem seus próprios identificadores numéricos, chamados TID (T de tupla). Os TIDs são os equivalentes RSS dos SRA (endereços dos registros armazenados) discutidos no Capítulo 2; são usados dentro do RSS para a construção de índices (e elos de ligação), aparecendo também ao RSI para que um “usuário” — normalmente codificação gerada na compilação pelo RDS — possa conseguir um acesso mais rápido a registros específicos (especificando os TIDs desses registros). *Não* aparecem para a SQL. A Figura 10.1 mostra como são implementados os TIDs.

O TID para um registro R consiste de duas partes: o número da página P que contém R e um deslocamento em bytes a partir do final de P identificando um espaço que, por sua vez, contém o deslocamento em bytes de R a partir do início de P. Este esquema representa um bom compromisso entre a rapidez do endereçamento direto e a flexibilidade do indireto; os registros podem ser rearranjados dentro de suas páginas — por exemplo para ocupar espaços vazios de registros removidos — sem que tenham que ocorrer mudanças nos TIDs (somente os deslocamentos locais do final da página terão que ser mudados); e o acesso a um registro cujo TID é conhecido é rápido, envolvendo somente um único acesso à página.²

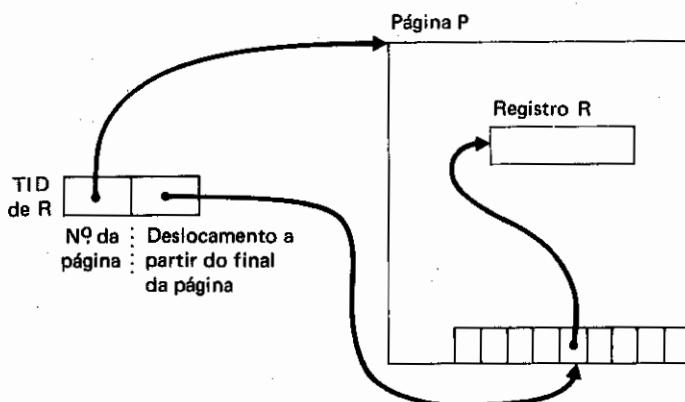


Fig. 10.1 Implementação de TIDs.

2

Em raras ocasiões pode envolver dois acessos a páginas (mas nunca mais de dois). Isto pode ocorrer se o registro ao ser atualizado tornar-se maior do que antes (por exemplo, por expansão do valor de algum campo de comprimento variável), e não houver espaço vazio suficiente na página para acomodar esse crescimento. Nessa situação, o registro atualizado é colocado em outra (“overflow”) página, sendo o registro original substituído por um indicador da nova localização. Se o mesmo voltar a ocorrer, de tal forma que o registro atualizado tenha que ser movido para uma terceira página, o registro indicador de localização na página original é mudado para indicar a mais nova localização.

Note, incidentalmente, que os RIDs e TIDs só são únicos dentro do segmento que os contém.

As operações RSI sobre registros incluem:

- **FETCH sid tid lista-de-campos lista-objetivo**

Os campos especificados do registro identificado por “sid” e “tid” são colocados nas localizações objetivo.

- **INSERT sid rid lista-fonte [tid]**

É criado um registro no arquivo especificado por “sid” e “rid”, obtendo valores das localizações fonte especificadas. É tentada a colocação do novo registro próximo ao registro identificado por “tid” (se especificado); assim é possível o arranjo de registros que são freqüentemente usados conjuntamente (por exemplo, departamento e seus empregados) para que sejam armazenados juntos. É retornado o TID do novo registro.

- **DELETE sid tid**

É removido o registro especificado por “sid” e “tid”.

- **UPDATE sid tid lista-de-campos lista-fonte**

Os campos especificados do registro identificado por “sid” e “tid” são atualizados, obtendo seus novos valores das localizações fonte especificadas.

10.4 CAMINHOS DE ACESSO

É sempre possível o acesso a um registro *diretamente* pelo seu TID, como explicado na Seção 10.3. Entretanto, às vezes é necessário ter-se acesso *seqüencialmente* aos registros (de acordo com alguma ordenação particular), ou diretamente por valor ao invés de por TID. O RSS fornece três tipos de caminhos de acesso para suportar essas necessidades: “seqüência do sistema”, índices, e *links*. Nesta seção discutiremos somente os casos de “sistema” e índices.

Seqüência do Sistema

Todo arquivo armazenado tem uma seqüência de sistema. Inicialmente (isto é, logo após a carga do arquivo), a seqüência do sistema é a “cronológica” — em outras palavras, é a seqüência com que os registros do arquivo foram criados. Entretanto, a seqüência do sistema irá aos poucos divergindo da cronológica, à medida que são feitas alterações (inserções e remoções) ao arquivo.

Para executar uma pesquisa exaustiva em um arquivo de acordo com a seqüência de sistema, o RSS pesquisará uma página de dados de cada vez (e não o índice de página) no segmento que contém o arquivo. Em cada uma dessas páginas, examinará cada um dos registros, inspecionando o prefixo para ver se estes pertencem ao arquivo em questão. (Uma determinada página pode conter registros de diversos arquivos diferentes.) Assim, cada página no segmento receberá exatamente um acesso na pesquisa. Em geral, a seqüência com que os registros são retornados parecerá randômica ao usuário.

Índices

Qualquer arquivo armazenado pode ter uma quantidade arbitrária de seqüências adicionais impostas por meio de índices apropriados. Um determinado índice define uma ordenação para o arquivo envolvido em termos dos valores de um ou mais campos daquele

arquivo. A seqüência ascendente ou descendente pode ser especificada separadamente para cada campo indexado dentro de um determinado índice. (É usado um esquema especial de codificação, de tal forma que, independentemente de quantos campos sejam indexados por um dado índice e de quantos desses campos tenham comprimento variável, o índice comporta-se como se fosse um campo único, de comprimento fixo [10.2]. A vantagem desse esquema é que ele aumenta em muito a velocidade da operação fundamental de comparação na qual são baseados todos os usos do índice.)

Internamente, os índices são organizados como árvores-B (veja o Capítulo 2), com cada bloco do índice ocupando uma página na memória (eis por que uma página pode ser de dados ou de índice, mas não de ambos). As páginas de índices encontram-se no mesmo segmento que as páginas de dados do arquivo indexado.

No RSI, os índices são identificados por um identificador numérico de índice (IID); o nome de índice especificado nas instruções SQL DDL CREATE INDEX e DROP INDEX é transformado em IID pelo RDS.

Para executar uma pesquisa exaustiva em um arquivo de acordo com um determinado índice, o RSS promove o acesso a todos os registros do arquivo na seqüência definida pelo índice. Como esta seqüência pode ser bastante diferente da seqüência de sistema, é possível que certas páginas de dados recebam vários acessos. (Por outro lado, páginas que não contenham qualquer registro do arquivo não receberão nenhum acesso, muito embora estejam no mesmo segmento.) Assim, uma pesquisa exaustiva via índice poderia potencialmente ser bem mais lenta do que uma pesquisa exaustiva usando a seqüência do sistema, *a menos que* o índice envolvido seja um “aglomerado”. Um índice aglomerado é aquele para o qual a seqüência definida pelo índice é a mesma (ou aproximadamente a mesma) seqüência do sistema. Um índice aglomerado só pode ser criado para um determinado arquivo armazenado no momento em que este é inicialmente carregado; além disso, os registros do arquivo têm que ser apresentados para a carga na seqüência correspondente ao índice (pois sua ordem de apresentação determina a seqüência inicial do sistema). Os índices aglomerados são importantes para o RDS no seu processo de otimização.

Um índice também pode fornecer acesso direto aos registros pelos valores de campos indexados. (A pesquisa seqüencial tem que ser usada para simular esta função, na falta de um índice adequado.) A operação RSI de *recuperação* direta por valor é como se segue.

- **FETCH sid rid iid lista-de-valores lista-de-campos lista-objetivo**

Os parâmetros “sid” e “rid” identificam um arquivo, e “iid” identifica um índice naquele arquivo (em certos campos). Os campos especificados na “lista-de-campos” são colocados nas localizações especificadas pela “lista-objetivo”, vindos dos registros especificados na “lista-de-valores” dos campos indexados.

As operações de DELETE e UPDATE pelo valor são definidas de forma análoga.

Varreduras

Vimos até agora como executar acesso *direto* por TID (Seção 10.3) e por valor (acima). Para executar um acesso *seqüencial*, precisamos de um outro objeto RSS — a varredura. As varreduras são similares aos cursores da SQL embutida; em outras palavras, elas fornecem um meio de movimentação através de um conjunto de registros, um de cada vez, em uma seqüência específica. A qualquer momento, uma varredura designa um *caminho de acesso* específico (por exemplo, um arquivo armazenado, em seqüência de sistema), e uma *posição* naquele caminho (por exemplo, um registro dentro do arquivo).

As varreduras são criadas e destruídas dinamicamente. Pode existir simultaneamente qualquer quantidade de varreduras. Uma varredura é identificada para o RSI por meio de um SCANID único. As operações envolvendo varreduras são como se segue.

- **OPEN** sid pathid [lista-de-valores]

O parâmetro “pathid” é ou um RID ou um IID. Uma varredura é criada para o arquivo (usando a seqüência de sistema) ou para o índice (usando a seqüência do índice), conforme for apropriado. No caso de varredura de um índice, podem ser opcionalmente especificados valores para os campos indexados. A varredura é posicionada no primeiro registro do caminho, ou no primeiro registro do caminho que possui os valores especificados nos campos indexados (se aplicável). O SCANID da nova varredura é retornado.

- **NEXT** scanid lista-de-campos lista-objetivo [predicado]

A varredura identificada por “scanid” é movida para o próximo registro ao longo do caminho que satisfaz ao predicado, e os campos especificados daquele registro são colocados nas localizações objetivo especificadas. (O predicado está restrito a uma combinação Booleana simples de comparações envolvendo campos do tipo de registro envolvido e variáveis do programa. Se o predicado for omitido, o efeito é como se tivesse sido especificado um predicado de *verdadeiro*.) Um parâmetro em NEXT especifica se a pesquisa deve começar no registro sobre o qual a varredura está correntemente posicionada ou no próximo registro ao longo do caminho.

- **DELETE** scanid

O registro identificado pelo “scanid” é removido.

- **UPDATE** scanid lista-de-campos lista-fonte

Os campos do registro identificado por “scanid” são atualizados, obtendo seus novos valores das localizações fonte especificadas.

- **CLOSE** scanid

A varredura identificada pelo “scanid” é destruída.

10.5 UM EXEMPLO

Apresentaremos um exemplo para esboçar como o RDS poderia fazer uso dos operadores RSS para implementar uma instrução SQL. Consideremos a consulta “Encontre os números dos fornecedores de Londres que fornecem a peça P1”, para a qual uma possível formulação SQL é

```
SELECT S#
  FROM  S
 WHERE CITY = 'LONDON'
   AND S# IN
        (SELECT S#
          FROM SP
         WHERE P# = 'P1')
```

Suponhamos que o banco de dados inclui os três seguintes índices (para maior clareza, usaremos identificadores simbólicos ao invés de numéricos):

- **XXS** (índice em fornecedores, por número de fornecedor);

- XSPS (índice em embarques, por número de fornecedor);
- XSPP (índice em embarques, por número de peça).

Mostraremos dois procedimentos possíveis para a implementação da consulta (suprimindo numerosos detalhes). O RDS irá escolher um destes procedimentos, ou talvez algum outro procedimento, atendendo a critérios como os de cardinalidade das tabelas e seu conhecimento sobre se algum dos índices é um índice aglomerado.

Procedimento 1 – Para cada fornecedor de Londres, verifique se há algum embarque da peça P1.

```

OPEN scan on file S;
let returned SCANID be SCAN1;
DO while more-to-come on SCAN1;
NEXT using SCAN1, CITY = 'LONDON'
      fetch S# into W1;
/* Insere a pesquisa na 1ª interação pelo registro "current" */ 
/* e pelo registro "next" nas interações subsequentes */ 
IF found THEN
DO;
  OPEN scan on index XSPS, S# = W1;
  let returned SCANID be SCAN2;
  NEXT using SCAN2, S# = W1, P# = 'P1';
  /* Inicie a pesquisa pelo registro "current" */
  IF found THEN output W1;
  CLOSE scan SCAN2;
END;
END;
CLOSE scan SCAN1;

```

Procedimento 2 – Para cada embarque de P1, verifique se o fornecedor está localizado em Londres.

```

OPEN scan on index XSPP, P# = 'P1';
let returned SCANID be SCAN3;
DO while more-to-come on SCAN3;
NEXT using SCAN3, P# = 'P1',
      fetch S# into W2;
/* Inicie a pesquisa na 1ª interação pelo registro "current" */ 
/* e pelo registro "next" nas interações subsequentes */ 
IF found THEN
DO;
  OPEN scan on index XSS, S# = W2;
  let returned SCANID be SCAN4;
  NEXT using SCAN4, S# = W2, CITY = 'LONDON';
  /* Inicie a pesquisa pelo registro "current" */
  IF found THEN output W2;
  CLOSE scan SCAN4;
END;
END;
CLOSE scan SCAN3;

```

10.6 O DIRETÓRIO RSS

Assim como o RDS mantém o dicionário do Sistema R para os objetos definidos no interface SQL (tabelas básicas, visões etc.), também o RSS mantém um diretório interno dos objetos definidos no RSI (arquivos armazenados, índices etc.). Usando este diretório, o RSS é capaz (por exemplo) de localizar todos os índices de um determinado arquivo armazenado, e ser assim capaz de garantir que as atualizações sobre aquele arquivo sejam refletidas naqueles índices.

O diretório está fisicamente distribuído através dos segmentos do banco de dados. Cada segmento inclui um conjunto de tabelas predefinidas (arquivos armazenados) contendo descrições do segmento e de todos os objetos dentro daquele segmento. São fornecidos operadores RSI especiais para obter, inserir, remover e atualizar os registros do diretório. Esses operadores (ou pelo menos os três últimos) são os análogos RSS das instruções DDL da SQL. Por exemplo, a criação de um novo índice envolve a inserção de registros apropriados de descrição no diretório; a remoção do índice envolve a remoção desses registros.

REFERÊNCIAS E BIBLIOGRAFIA

A maior parte do material deste capítulo foi obtida em [5.1] e [5.3].

10.1 R. A. Lorie. "Physical Integrity in a Large Segmented Database." *ACM Transactions on Database Systems* 2, nº 1 (março de 1977).

Este artigo está voltado para o Componente de Armazenamento do RSS. Este componente é o responsável pelo manuseio dos dispositivos físicos, dando aos outros componentes uma visão do banco de dados em termos de segmentos e páginas. Como o título indica, o artigo ocupa-se basicamente das técnicas de recuperação do banco de dados após falhas (quando a falha danifica ou o próprio banco de dados ou o *buffer* na memória principal). Entretanto, o artigo também fornece detalhes sobre segmentos e páginas em si.

10.2 M. W. Blasgen, R. G. Casey, and K. P. Eswaran. "An Encoding Method for Multifield Sorting and Indexing." *CACM* 20, nº 11 (novembro de 1977).

10.3 D. Bjørner, E. F. Codd, K. L. Deckert, and I. L. Traiger. "The GAMMA-O *n*-ary Relational Data Base Interface: Specifications of Objects and Operations." IBM Research Report RJ1200 (abril de 1972).

GAMMA-O foi um interface hipotético de baixo nível para um sistema relacional (pretendia-se que fosse usado na implementação de uma linguagem de nível mais alto, tal como a SQL). Embora nunca tenha sido implementado, muitas de suas idéias foram incorporadas no RSI.

10.4 R. A. Lorie. "XRM – An Extended (*n*-ary) Relational Memory." IBM Technical Raport G320-2096 (janeiro de 1974).

XRM (outra antecessora do RSS) foi usada como método de acesso do protótipo original do SEQUEL [5.7, 5.8]. XRM, por sua vez, foi desenvolvida para um sistema anterior chamado RAM [10.5].

10.5 M. F. Crick and A. J. Symonds. "A Software Associative Memory for Complex Data Structures." IBM Technical Report G320-2060 (1972).

11

Query By Example

11.1 INTRODUÇÃO

Neste capítulo, vamos examinar outro sistema relacional, o Query By Example (originalmente desenvolvido por M. M. Zloof no Laboratório de Pesquisas IBM de Yorktown Heights [11.3–11.9], e agora disponível na IBM como um “Installed User Program” correndo sob o sistema operacional VM/CMS [11.1, 11.2]. O nome “Query By Example” (ou QBE) é usado tanto para designar o sistema em si como a linguagem que o sistema suporta, e nós seguiremos essa prática, deixando que o contexto esclareça o que queremos significar. O nível funcional do QBE é aproximadamente o mesmo da SQL. Mas o QBE difere da SQL por ter sido projetado especificamente para uso com terminais visuais de tela; não somente os resultados são apresentados na forma de tabelas (como no User-Friendly Interface do Sistema R), como também todas as solicitações do usuário são especificadas pelo preenchimento de tabelas na tela. Essas tabelas são construídas parcialmente pelo sistema e parcialmente pelo usuário (examinaremos esse processo com mais alguns detalhes adiante). Como as operações são especificadas em forma tabular, dizemos que o QBE tem uma sintaxe *bidimensional*. Em contraste, a maioria das linguagens tradicionais de computação tem uma sintaxe linear.¹

Como a SQL, o QBE passou por uma certa quantidade de testes de uso, com resultados muito encorajadores [11.11, 11.12].

O nome “Query By Example” deriva do fato de serem usados *exemplos* na especificação das consultas (e também na maioria das outras operações; por enquanto vamos considerar somente a recuperação). A idéia básica é a de que o usuário formule a consulta entrando com um exemplo de resposta possível no local apropriado de uma tabela vazia. Por exemplo, consideremos a consulta “Obtenha os números dos fornecedores de Paris”.

¹ Embora objetivando inicialmente o uso interativo, o QBE pode ser invocado por uma chamada via rotina de interface pelo PL/I ou pelo APL. Com esta finalidade, foi desenvolvida uma versão linear da sintaxe QBE [11.2]. Mas a sintaxe linear não é tão atraente quanto a forma bidimensional.

Primeiro, apertando uma determinada *tecla funcional* do terminal, o usuário pode ter projetada na tela do terminal uma tabela “esqueleto” em branco. Depois, sabendo que a resposta à consulta pode ser encontrada na tabela S, o usuário entra com S como nome da tabela e, de uma forma a ser descrita (Exemplo 11.6.2), consegue que o QBE responda preenchendo os nomes de colunas de S. Agora o usuário pode expressar a consulta colocando entradas em duas posições desta tabelas, como se segue.

S	S#	SNAME	STATUS	CITY
	P. <u>S7</u>			PARIS

O “P.” significa “imprimir” (*Print*); ele indica o objetivo da consulta, isto é, os valores que devem aparecer no resultado. S7 é um “elemento de exemplo”, isto é, um exemplo de possível resposta à consulta; elementos de exemplo são sublinhados. PARIS (não sublinhado) é um “elemento constante”. A consulta pode ser parafraseada: “Imprima todos os valores S#, tal como S7 (digamos), em que a cidade correspondente seja Paris.” Observe que S7 não precisa realmente aparecer no conjunto resultante, ou mesmo no conjunto original; o elemento de exemplo é completamente arbitrário, e poderíamos ter da mesma forma usado *PIG*, ou 7, ou *X*, sem mudar o significado da consulta.²

Mais tarde veremos que os elementos de exemplo são usados para estabelecer ligações entre colunas em consultas mais complicadas. Se não forem necessárias ligações, como na consulta acima, é possível omitir-se inteiramente elementos de exemplo (e assim “P.S7” se reduziria somente a “P.”), mas para maior clareza nós normalmente os incluímos.

Vamos agora apresentar os mais importantes dispositivos da linguagem QBE por meio de uma série de exemplos. Trataremos das operações DML antes das operações DDL porque, como veremos, as operações DDL são essencialmente casos especiais das operações DML. A linguagem como originalmente definida [11.3] inclui certos dispositivos que não estão suportados na implementação da IBM; vamos ignorar a maior parte desses dispositivos neste capítulo. (Chamamos a atenção de que a implementação da IBM inclui alguns dispositivos adicionais, especialmente no domínio de autorizações, que também não discutiremos aqui.)

11.2 OPERAÇÕES DE RECUPERAÇÃO

11.2.1 Recuperação simples – Obtenha os números de todas as peças fornecidas. (Exemplo 7.2.1)

SP	S#	P#	QTY
		P. <u>PX</u>	

Diferentemente da SQL, o QBE elimina as duplicações redundantes de um resultado de consulta. Para suprimir essa eliminação, o usuário pode especificar a palavra-chave “ALL”.

² A implementação do QBE da IBM usa um caractere único de sublinhado como prefixo, ao invés de sublinhar o elemento de exemplo para indicá-lo (por exemplo, S7 aparece como _S7).

(Os operadores QBE sempre terminam com um ponto.) Por exemplo,

SP	S#	P#	QTY
	P.ALL.PX		

Incidentalmente, se uma coluna for muito estreita na tela para conter a entrada desejada, o usuário pode alargá-la primeiro por meio de outra tecla funcional. Há também teclas de função para adição de colunas, adição de linhas, remoção de colunas, remoção de linhas e assim por diante.

11.2.2 Recuperação simples – Obtenha todos os detalhes de todos os fornecedores. (Exemplo 7.2.2)

S	S#	SNAME	STATUS	CITY
	P.SX	P.SN	P.ST	P.SC

A representação a seguir é uma forma resumida da mesma consulta

S	S#	SNAME	STATUS	CITY
P.				

Aqui, o operador de impressão está aplicado a toda a linha.

11.2.3 Recuperação qualificada – Obtenha os números dos fornecedores de Paris com status > 20. (Exemplo 7.2.3).

S	S#	SNAME	STATUS	CITY
	P.SX		> 20	PARIS

Observe como a condição “status > 20” está especificada. Em geral, qualquer dos operadores de comparação $=$, \neq , $<$, $<=$, $>$, \geq pode ser usado desta forma (exceto que $=$ é normalmente omitido, como na coluna CITY acima, e \neq é usualmente abreviado simplesmente como \neg). Também, qualquer desses operadores pode ser precedido por “P.” caso desejemos imprimir o valor envolvido. Por exemplo, poderíamos ter especificado “P. > 20” sob STATUS acima, caso quiséssemos imprimir também os valores de status.

11.2.4 Recuperação qualificada – Obtenha os números dos fornecedores localizados em Paris ou que tenham status > 20 (ou ambos).

S	S#	SNAME	STATUS	CITY
	P.SX			PARIS
	P.SY		> 20	

As condições especificadas dentro de uma mesma linha são consideradas “E” (AND). Para que duas condições sejam portanto “OU” (OR), é necessário especificá-las em linhas separadas. A consulta acima está efetivamente solicitando a *união* de todos os números *SX* dos fornecedores de Paris e todos os números *SY* dos fornecedores com status > 20. São necessários dois elementos de exemplo diferentes, pois se tivéssemos usado o mesmo duas vezes, isto teria significado que o *mesmo* fornecedor teria que estar em Paris e ter status > 20.

Quando uma consulta envolve mais de uma linha, como neste exemplo, as linhas podem ser entradas em qualquer ordem.

11.2.5 Recuperação qualificada – Obtenha os números dos fornecedores que fornecem tanto a peça P1 como a peça P2.

SP	S#	P#	QTY
	P. <u>SX</u>	P1	
	<u>SX</u>	P2	

Aqui, o mesmo elemento de exemplo *tem* que ser usado duas vezes; nós precisamos de duas linhas para expressar a consulta porque é necessário estabelecer o “E” (AND) entre duas condições na mesma coluna.

11.2.6 Recuperação com ordenação – Obtenha os números e os status dos fornecedores de Paris, em ordem descendente de status. (Exemplo 7.2.4)

S	S#	SNAME	STATUS	CITY
	P. <u>SX</u>		P.DO. <u>ST</u>	PARIS

O “DO.” significa “ordem descendente”. “AO.” é usado para ordenação ascendente. Quando é necessária ordenação em termos de diversos campos, então o campo maior é indicado por “AO(1).” [ou “DO(1).”], o próximo por “AO(2).” [ou “DO(2).”] e assim por diante.

11.2.7 Recuperação usando uma ligação – Obtenha os nomes dos fornecedores que fornecem a peça P2 (Exemplos 7.2.7, 7.29, e 7.2.16).

S	S#	SNAME	STATUS	CITY
	<u>SX</u>	P. <u>SN</u>		

SP	S#	P#	QTY
	<u>SX</u>	P2	

O elemento exemplo *SX* é usado como uma ligação entre *S* e *SP*. A consulta pode ser parafraseada: “Imprima os nomes dos fornecedores (como *SN*) onde o correspondente número de fornecedor, digamos *SX*, apareça na tabela *SP* casado com o número de peça *P2*”. De forma genérica, as ligações são usadas em QBE onde a SQL usaria um ninho de SELECT ou um quantificador existencial, ou onde a álgebra usaria uma união. De fato, *SX* no exemplo 11.2.5 também estava atuando como uma ligação, mas as linhas sendo ligadas estavam na mesma tabela.

11.2.8 Recuperação usando ligações – Obtenha os nomes dos fornecedores que fornecem pelo menos uma peça vermelha. (Exemplo 7.2.10)

S	S#	SNAME	STATUS	CITY
	<u>SX</u>	P.SN		

SP	S#	P#	QTY
		<u>PX</u>	

P	P#	PNAME	COLOR	WEIGHT	CITY
	<u>PX</u>		RED		

11.2.9 Recuperação quando negação – Obtenha os nomes dos fornecedores que não fornecem a peça P2. (Exemplos 7.2.14 e 7.2.17)

S	S#	SNAME	STATUS	CITY
	<u>SX</u>	P.SN		

SP	S#	P#	QTY
¬	<u>SX</u>	P2	

Observe o operador NOT (\neg) sobre a linha de consulta na tabela SP. A consulta pode ser parafraseada: “Imprima os nomes dos fornecedores *SX* de tal forma que *SX* não forneça a peça P2.”

11.2.10 Recuperação usando uma ligação dentro de tabela singela – Obtenha os números dos fornecedores que fornecem pelo menos uma peça fornecida pelo fornecedor S2. (Exemplo 7.2.12; compare também com o exemplo 11.2.5.)

SP	S#	P#	QTY
	<u>P.SX</u>	<u>PX</u>	
	S2	<u>PX</u>	

11.2.11 Recuperação usando uma ligação dentro de tabela singela – Obtenha os números de todas as peças fornecidas por mais de um fornecedor. (Exemplos 7.2.13 e 7.3.7)

SP	S#	P#	QTY
	<u>SX</u>	<u>P.PX</u>	
¬	<u>SX</u>	<u>PX</u>	

Esta consulta pode ser parafraseada: “Imprima os números das peças *PX* de tal forma que *PX* seja fornecida por algum fornecedor *SX* e também por algum fornecedor distinto de *SX*.”

11.2.12 Recuperação de mais de uma tabela – Para cada peça fornecida, obtenha os números das peças e os nomes de todas as cidades que fornecem a peça. (Exemplo 7.2.5) O resultado desta consulta não é uma projeção de alguma tabela existente; é a projeção de uma junção de duas tabelas existentes. Para formular esta consulta em Query By

Example, o usuário tem primeiramente que criar uma tabela esqueleto com o mesmo formato do resultado esperado (isto é, com o número apropriado de colunas). Esta tabela e suas colunas podem receber quaisquer nomes que o usuário deseje – podem até serem deixadas sem nome. O usuário pode então expressar a consulta usando a tabela “resultado” e as duas tabelas existentes, como se segue.

S	S#	SNAME	STATUS	CITY	SP	S#	P#	QTY
						SX	PX	
				<u>SC</u>				

RESULT	P#	CITY
	P.PX	P.SC

11.2.13 Recuperação usando o bloco de condição – Obtenha todos os pares de números de fornecedores de forma a que os dois fornecedores estejam localizados na mesma cidade. (Exemplo 7.2.6)

S	S#	SNAME	STATUS	CITY	RESULT	FIRST	SECOND
					P.	SX	SY
	SX			<u>CZ</u>			
	SY			<u>CZ</u>			

CONDITIONS
<u>SX < SY</u>

Ocasionalmente torna-se difícil expressar alguma condição desejada dentro da estrutura da tabela de consulta. Nessa situação, o QBE permite que o usuário entre com a condição em um “bloco de condição” separado, como este exemplo ilustra. O bloco de condição é obtido usando-se outra tecla funcional do terminal. (Lembramos ao leitor que a razão para exigir que o número do primeiro fornecedor seja menor do que o do segundo é a de eliminar pares da forma (x, x) e garantir que no máximo um dos pares (x, y), (y, x) apareça.)

11.3 OPERAÇÕES DE RECUPERAÇÃO EM RELAÇÕES DE ESTRUTURA EM ÁRVORE

As operações QBE descritas nesta seção não estão suportadas na implementação corrente da IBM. No entanto, elas merecem um exame, pois endereçam uma área interessante de aplicação, a qual poucas linguagens, relacionais ou não, endereçam com elegância. Maiores detalhes sobre essas operações podem ser encontrados em [11.5].

Consideremos a relação RS (estrutura de subordinação) mostrada na Fig. 11.1. Esta relação possui dois atributos, EMP# e MGR#, ambos obtidos de um domínio básico de números de empregados. O significado de uma determinada tupla de RS é que o empregado indicado (EMP#) se subordina diretamente ao gerente indicado (MGR#).

RS	MGR#	EMP#
E1	E6	
E1	E7	
E1	E8	
E6	E18	
E8	E15	
E8	E16	
E15	E20	
E15	E24	
E24	E32	

Fig. 11.1 Relação RS de estrutura de subordinação.

Estamos supondo que a estrutura de subordinação representada pela relação RS satisfaz às seguintes restrições (todo o tempo).

1. Nenhum empregado é seu próprio gerente.
2. Nenhum empregado tem mais do que um gerente imediato.
3. Se EX (por exemplo) for o gerente imediato de EY , então EY não pode ser gerente de EX em nenhum nível.

O leitor pode observar que a tabulação de exemplo da Fig. 11.1 não satisfaz às restrições. Devido a esse fato, podemos representar a relação como uma estrutura em árvore, como na Fig. 11.2.

RS	MGR#	EMP#
	E8	P.EX

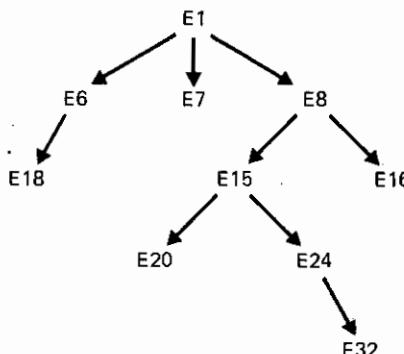


Fig. 11.2 Relação RS representada como uma árvore.

Nós iremos nos referir a uma relação possuindo dois atributos definidos em um domínio comum e satisfazendo a restrições análogas às acima como sendo uma *relação de estrutura em árvore*. No Query By Example, o usuário está capacitado a formular certas consultas como uma relação estruturada em árvore, que linguagens de menor potência tais como a álgebra relacional são incapazes de expressar. Vamos considerar alguns exemplos.

11.3.1 Recuperação descendo um nível — Obtenha os números dos empregados que se subordinam ao empregado E8 no primeiro nível.

Por “no primeiro nível” queremos significar que E8 é o gerente imediato dos empregados que desejamos. A resposta à consulta é — empregados E15 e E16. Este exemplo é imediato e não ilustra itens novos.

11.3.2 Recuperação descendo dois níveis — Obtenha os números dos empregados que se subordinam ao empregado E8 em segundo nível.

RS	MGR#	EMP#
	E8	EY
	EY	P.EX

Novamente a solução é direta — mas observe que tivemos que introduzir uma ligação, EY, e tivemos que entrar com essa ligação na tabela duas vezes. Em geral, se quisermos descer n níveis na árvore, teremos que entrar com cada um dos $n - 1$ índices duas vezes, um processo bastante tedioso. O Query By Example fornece um caminho mais curto conveniente, que está ilustrado na seguinte formulação alternativa da consulta acima.

RS	MGR#	EMP#
	E8	P.EX(2L)

O “2L” significa “segundo nível”. Em geral, uma entrada de nível pode consistir de qualquer inteiro seguido pela letra L, tudo colocado entre parêntesis. Sempre que é usado um nível de entrada, o Query By Example inclui níveis relativos no resultado tabulado, como no exemplo que se segue.

EMP#
E20(2L)
E24(2L)

11.3.3 Recuperação subindo dois níveis — Obtenha o número de empregado do gerente dois níveis acima do empregado E20.

RS	MGR#	EMP#
	P.MX(2L)	E20

Aqui o nível de entrada aparece na coluna MGR#. Em geral, a direção da pesquisa (subindo ou descendo a árvore) é indicada pela coluna na qual aparece o nível de entrada. Em certas situações, esta regra poderia levar entretanto a uma ambigüidade [11.5]; para evitar este problema, o Query By Example impõe uma restrição de que não mais do que duas entradas podem aparecer em uma determinada linha, na formulação de uma consulta que envolva níveis.

11.3.4 Recuperação descendendo a todos os níveis — Obtenha os números dos empregados que se subordinam ao empregado E8 em *qualquer* nível.

RS	MGR#	EMP#
	E8	P. <u>EX(6L)</u>

Resultado:

EMP#
E15(1L)
E20(2L)
E24(2L)
E32(3L)
E16(1L)

Este é um exemplo de consulta que não pode ser expressa na álgebra relacional ou em linguagens de potência equivalente. Observe o sublinhado no nível de entrada.

11.3.5 Recuperação descendendo ao nível mais baixo — Obtenha os números dos empregados que se subordinam ao empregado E8 no nível *mais baixo*.

RS	MGR#	EMP#
	E8	P. <u>EX(MAX.6L)</u>

MAX é uma função integrada (veja Seção 11.4). O significado desta consulta é “Obtenha os números dos empregados cujo nível relativo abaixo de E8 é o maior”. O resultado é um único empregado — E32 (3L).

11.3.6 Recuperação descendendo aos níveis terminais — Obtenha os números dos empregados que se subordinam ao empregado E8 e que não têm ninguém se subordinando a eles.

RS	MGR#	EMP#
	E8	P. <u>EX(LAST.L)</u>

Estamos procurando empregados nos nós terminais da árvore sob E8. Como esses empre-

gados estarão em níveis relativos diferentes, em geral, não podemos entrar nem com uma constante inteira nem com um inteiro de exemplo (uma constante significaria um nível fixo, e um exemplo significaria todos os níveis); por isso, o Query By Example fornece a função especial LAST.

11.3.7 Recuperação de nível – A que nível está o empregado E20 abaixo do empregado E1?

RS	MGR#	EMP#
E1	P.E20(7L)	

Resultado:

EMP#
E20(3L)

11.4 FUNÇÕES INTEGRADAS

Tal como a SQL, o QBE oferece um conjunto de funções integradas, como se segue.

CNT.ALL. CNT.UNQ.ALL.
SUM.ALL. SUM.UNQ.ALL.
AVG.ALL. AVG.UNQ.ALL.
MAX.ALL.
MIN.ALL.

Observe que “ALL.” está sempre especificado. O “UNQ.” opcional significa “elimine as duplicações redundantes antes de aplicar a função” (somente em CNT., SUM., e AVG.). Os significados das outras palavras-chave são óbvios. Os valores nulos são sempre eliminados, exceto para CNT.

11.4.1 Recuperação simples usando uma função – Obtenha a quantidade total de fornecedores. (Exemplo 7.3.1)

S	S#	SNAME	STATUS	CITY
P.CNT.ALL.SX				

11.4.2 Recuperação simples usando uma função – Obtenha a quantidade total de fornecedores correntemente fornecendo peças. (Exemplo 7.3.2)

SP	S#	P#	QTY
P.CNT.UNQ.ALL.SX			

11.4.3 Recuperação qualificada usando uma função – Obtenha a quantidade de embarques da peça P2. (Exemplo 7.3.3)

SP	S#	P#	QTY
	P.CNT.ALL.SX	P2	

11.4.4 Recuperação qualificada usando uma função – Obtenha a quantidade total de peças P2 fornecidas. (Exemplo 7.3.4)

SP	S#	P#	QTY
		P2	P.SUM.ALL.Q

11.4.5 Função em um bloco de condição – Obtenha os números dos fornecedores cujo valor de status seja menor do que o máximo valor de status na tabela S (Exemplos 7.2.8 e 7.3.5); obtenha também aquele valor máximo.

S	S#	SNAME	STATUS	CITY	CONDITIONS
	P.SX		ST P.MAX.ALL.SS		ST < MAX.ALL.SS

11.4.6 Recuperação com grupamento – Para cada peça fornecida, obtenha o número da peça e a quantidade total fornecida daquela peça. (Exemplo 7.3.6)

SP	S#	P#	QTY
		P.G.PX	P.SUM.ALL.QX

O “G.” no QBE é equivalente ao operador GROUP BY da SQL.

11.4.7 Recuperação usando grupamento e o bloco de condição – Obtenha os números de todas as peças fornecidas por mais de um fornecedor (mesmo que o exemplo 11.2.11); obtenha também as quantidades dos fornecedores correspondentes.

SP	S#	P#	QTY	CONDITIONS
	P.CNT.ALL.SX	P.G.PX		CNT.ALL.SX > 1

11.5 OPERAÇÕES DE ATUALIZAÇÃO

11.5.1 Atualização de registro único Mude a cor da peça P2 para amarela.

P	P#	PNAME	COLOR	WEIGHT	CITY
	P2		U. YELLOW		

Ou:

P	P#	PNAME	COLOR	WEIGHT	CITY
U.	P2		YELLOW		

A operação de atualização é “U.” O registro a ser atualizado é identificado por sua chave primária. Os valores das chaves primárias não podem ser atualizados.

11.5.2 Atualização de registro único baseada em tabela anterior – Aumente o peso da peça P2 de 5.

P	P#	PNAME	COLOR	WEIGHT	CITY
	P2			WT	
U.	P2			WT + 5	

Para atualizar um registro com base no seu valor antigo, o usuário dá entrada em uma linha representando a versão antiga, e em outra linha representando a nova versão. O “U.” indica qual é a nova.

11.5.3 Atualização de múltiplos registros – Duplique o status de todos os fornecedores de Londres. (Exemplo 7.4.2)

S	S#	SNAME	STATUS	CITY
	SX		ST	LONDON
U.	SX		2 * ST	

Aqui, o valor da chave primária (número do fornecedor) está especificado como um elemento de exemplo, não como uma constante.

11.5.4 Atualização de múltiplos registros – Ajuste para zero as quantidades de todos os fornecedores de Londres. (Exemplo 7.4.3)

SP	S#	P#	QTY	S	S#	SNAME	STATUS	CITY
	SX		0		SX			LONDON

11.5.5 Atualização de múltiplas tabelas — Ajuste para zero as quantidades e status de todos os fornecedores de Londres.

SP	S#	P#	QTY
U.	<u>SX</u>		0

S	S#	SNAME	STATUS	CITY
	<u>SX</u>			LONDON
	<u>SY</u>			LONDON
U.	<u>SY</u>		0	

Uma vez que a unidade de entrada para o QBE é toda a tela, diversas atualizações podem ser entradas simultaneamente.

11.5.6 Inserção de registro único — Adicione a peça P7 (nome ‘WASHER’, cor ‘GREY’, peso 2, cidade ‘ATHENS’) à tabela P. (Exemplo 7.4.5)

P	P#	PNAME	COLOR	WEIGHT	CITY
I.	P7	WASHER	GREY	2	ATHENS

“I.” é o operador de inserção. O novo registro tem que ter um valor de chave primária não nulo e distinto de todos os outros valores existentes de chaves primárias na tabela. Podem ser deixados em branco outros campos na tabela esqueleto, e nesse caso eles terão seus valores nulos no banco de dados.

11.5.7 Inserção de múltiplos registros — A tabela TEMP possui uma coluna denominada P#. Dê entrada em TEMP de todos os números das peças fornecidas pelo fornecedor S2 (veja Exemplo 7.4.6).

TEMP	P#
I.	<u>PX</u>

SP	S#	P#	QTY
	S2	<u>PX</u>	

11.5.8 Remoção de registro único — Remova o fornecedor S1. (Exemplo 7.4.7)

S	S#	SNAME	STATUS	CITY
D.	S1			

“D.” é o operador de remoção. Se a tabela SP correntemente tiver algum embarque do fornecedor S1, esta remoção causará violação da Regra de Integridade 2. (Mas veja o Exemplo 11.5.11.)

11.5.9 Remoção de múltiplos registros – Remova todos os fornecedores de Londres.

S	S#	SNAME	STATUS	CITY
D.				LONDON

Novamente esta remoção pode causar violação da Regra de Integridade 2.

11.5.10 Remoção de múltiplos registros – Remova todos os embarques (Exemplos 7.4.8)

SP	S#	P#	QTY
D.			

11.5.11 Remoção de múltiplos registros, múltiplas tabelas – Remova todos os fornecedores de Londres e também todos os embarques daqueles fornecedores. (Exemplo 7.4.9)

S	S#	SNAME	STATUS	CITY	SP	S#	P#	QTY
D.	<u>SX</u>			LONDON	D.	<u>SX</u>		
				LONDON				

Neste Exemplo, o QBE removerá primeiramente os embarques e depois os fornecedores.

11.6 O DICIONÁRIO QBE

Tal como no Sistema R, o QBE possui um dicionário próprio integrado, que é representado como uma coleção de tabelas para o usuário. O dicionário inclui, por exemplo, uma tabela TABLE e uma tabela DOMAIN, dando detalhes sobre todas as tabelas e todos os domínios correntemente conhecidos pelo sistema. Como no Sistema R, as tabelas do dicionário podem ser interrogadas usando-se operadores ordinários de recuperação da DML. No entanto, o QBE vai um pouco mais além do que o Sistema R nesse aspecto, pois os operadores para consulta e *atualização* do dicionário estão integrados à linguagem, de uma forma consistente com os outros operadores (não relacionados ao dicionário). Em particular, o QBE não inclui realmente uma DDL em si, mas sim usa formas especiais dos operadores de atualização DML para fornecer a função equivalente.

Começaremos mostrando dois exemplos de recuperação.

11.6.1 Recuperação de nomes de tabelas – Obtenha os nomes de todas as tabelas conhecidas pelo sistema.

P.				

Ao invés de ter que construir um esqueleto da tabela TABLE e entrar “P.” na coluna NAME daquele esqueleto, o usuário tem que apenas entrar com “P.” na posição de nome de tabela de uma tabela em branco.

11.6.2 Recuperação de nomes de colunas de determinada tabela — Obtenha os nomes de todas as colunas da tabela S.

S	P.				

Para formular esta consulta, o usuário entra com o nome da tabela (s) seguido de “P.” na linha (em branco) de nomes de colunas. O QBE responde preenchendo adequadamente os brancos. Esta função é comumente usada no preparo de uma consulta “real”. (Nota: Se o resultado contiver nomes de colunas que não sejam requeridos para expressar a consulta “real”, esses nomes poderão ser eliminados pelo uso de uma tecla funcional, antes que o usuário dê prosseguimento.)

Vamos agora observar as funções de definição de dados.

11.6.3 Criação de uma nova tabela — Crie a tabela S (supondo que ela ainda não existe).

I.	S I.	S#	SNAME	STATUS	CITY

O primeiro “I.” cria uma entrada no dicionário para a tabela S; o segundo “I.” cria entradas no dicionário para as quatro colunas da tabela S. No entanto, o processo de criação da tabela ainda não está completo; têm que ser especificadas algumas informações adicionais para cada coluna. Essas informações adicionais incluem (para cada coluna) — o nome do domínio básico; o tipo de dado do domínio, se este ainda não é conhecido pelo QBE; uma indicação de se a coluna participa ou não da chave primária; e uma especificação sobre se deve ou não ser construído um índice (“inversão”) na coluna. (O QBE assume que toda coluna faz parte da chave, e que toda coluna deve ser indexada, a menos que seja informado o contrário.) Mostraremos um esboço de como esta informação pode ser especificada para a tabela S; para maiores detalhes do processo de especificação e de outras informações a serem especificadas, o leitor deverá procurar [11.2].

S	S#	SNAME	STATUS	CITY
DOMAIN	I.	S#	SNAME	STATUS
TYPE	I.	CHAR (5)	CHAR (20)	FIXED
KEY		Y	U.N	U.N
INVERSION		Y	U.N	U.N

Neste exemplo, as informações sobre DOMAIN (domínio) e TYPE (tipo) foram fornecidas pela inserção de uma linha completa de especificações; as informações fornecidas sobre KEY (chave) e INVERSION (inversão) atualizam os valores intrínsecos do QBE. Observe que nós agora definimos *domínios* para o QBE, bem como a tabela S e colunas; por exemplo, CITY é agora conhecido pelo QBE como sendo um domínio de seqüência de caracteres de comprimento 15. Os tipos de dados suportados pelo QBE são CHAR (isto é, seqüência de caracteres de comprimento variável), CHAR (*n*), FIXED, FLOAT, DATE e TIME.

11.6.4 Criando um instantâneo — Para cada peça fornecida, obtenha o número da peça e os nomes de todas as cidades que a fornecem (Exemplo 11.2.12), e guarde o resultado no banco de dados.

S	S#	SNAME	STATUS	CITY	SP	S#	P#	QTY
	<u>SX</u>			<u>SC</u>		<u>SX</u>	<u>PX</u>	
<hr/>								
I. RESULT		I.	P#	CITY				
P.I.			<u>PX</u>	<u>SC</u>				

RESULT é avaliado como no Exemplo 11.2.12 e apresentado na tela (devido ao "P." da última linha); além disso, uma cópia dessa tabela com o nome de RESULT e nomes de colunas P# e CITY é armazenada no banco de dados (devido aos operadores "I."). As especificações de domínio (e portanto os tipos de dados) das colunas desta tabela são herdados das tabelas básicas; outras especificações (por exemplo KEY, INVERSION) são obtidas intrinsecamente (mas podem ser modificadas, se desejado, usando-se "U."); entretanto, essas mudanças só podem ser especificadas no momento de criação da tabela, não em operações subsequentes). A nova tabela armazenada é dita ser um *instantâneo* das tabelas básicas. Uma vez criada, ela se torna independente daquelas tabelas — isto é, as tabelas básicas e o instantâneo podem ser atualizados independentemente; o instantâneo não é uma visão (no sentido do Sistema R).

11.6.5 Destruindo uma tabela — Destrua a tabela SP.

Uma tabela só pode ser destruída se estiver correntemente vazia.

Etapa 1: Remova todos os embarques. (Exemplo 11.5.10)

SP	S#	P#	QTY
D.			

Etapa 2: Destrua a tabela.

D. SP	S#	P#	QTY

11.6.6 Expandindo uma tabela – Adicione uma coluna DATE à tabela SP. Diferentemente do Sistema R, o QBE não suporta diretamente a adição dinâmica de uma nova coluna a uma tabela existente, a menos que essa tabela esteja correntemente vazia. Por isso é necessário fazer-se o seguinte:

1. Definir uma nova tabela com o mesmo formato da existente mais a nova coluna.
2. Carregar a nova tabela a partir da antiga, usando uma inserção de múltiplos registros.
3. Remover todos os dados da tabela antiga.
4. Destruir a tabela antiga.
5. Mudar o nome da nova tabela para o que designava a antiga.

Etapas 1 e 2:

SP	S#	P#	QTY
I.	SX	PX	QX

I.	SPCOPY	I.	S#	P#	QTY	DATE
	DOMAIN					U. DATES
	TYPE					U. DATE
	KEY				U.N	U.N
	INVERSION				U.N	U.N
I.			SX	PX	QX	

Etapa 3:

SP	S#	P#	QTY
D.			

Etapa 4:

D.	SP	S#	P#	QTY

Etapa 5:

SPCOPY	U. SP	S#	P#	QTY	DATE

Todos os valores DATE na nova tabela SP estarão inicialmente nulos.

11.7 DISCUSSÃO

Uma vantagem significativa do QBE, quando comparado à maioria das outras linguagens de consulta, é o grau de *liberdade* de que goza o usuário – isto é, liberdade para elaborar

a consulta da maneira que lhe parecer mais natural. Especificamente, a consulta pode ser preparada na *ordem* que o usuário quiser; a ordem das colunas dentro de uma tabela de consulta é inteiramente imaterial. Além disso, a ordem em que o usuário preenche todas as entradas que constituem as linhas da tabela é também completamente arbitrária. Por exemplo, veja o exemplo 11.2.8 (“Obtenha os nomes dos fornecedores que fornecem peças vermelhas”). O usuário pode imaginar esta consulta como: “Separe as peças vermelhas, depois os números dos fornecedores dessas peças, e depois os nomes correspondentes” — caso em que o usuário provavelmente completará as tabelas de consulta na ordem P, SP, S. Alternativamente, o usuário poderá imaginar assim: “Separe nomes de fornecedores tais que os fornecedores correspondentes supram uma peça que seja vermelha” — neste caso, o usuário provavelmente preencherá as tabelas na ordem S, SP, P. De qualquer forma, a consulta final é a mesma. Em outras palavras, o Query By Example é uma linguagem altamente não procedural, capaz de suportar diversas formas diferentes de se imaginar um problema; não força a que todos os usuários percebam o problema da mesma maneira.

Por ser de interesse, ofereceremos algumas comparações funcionais entre o produto QBE e o Sistema R. (Naturalmente, a maioria dessas comparações reflete características das implementações, e não propriedades intrínsecas das linguagens QBE e SQL.)

- *Compartilhamento dinâmico* — Um banco de dados QBE pode ser compartilhado por qualquer quantidade de usuários que façam somente leitura de dados, *ou* ser dedicado a um único usuário fazendo atualização. Em contraste, no Sistema R o banco de dados fica normalmente disponível a qualquer quantidade simultânea de atualizações.
- *Definição dinâmica do banco de dados* — O QBE não é tão flexível como o Sistema R nesse aspecto. Veja o Exemplo 11.6.6.
- *Visões* — Embora a linguagem completa do QBE inclua visões, (no sentido do Sistema R), estas não estão suportadas na implementação corrente da IBM.
- *Compilação x interpretação* — O QBE usa a abordagem interpretativa convencional. Não há pré-compilador QBE.
- *Interface com o programa de aplicação* — Tal como a SQL, o QBE está disponível tanto para uso em terminais *on-line* como para programas de aplicação *batch*. No entanto, o interface para “QBE embutido” é bastante rudimentar.
- *Estrutura de dados* — As tabelas QBE são realmente relações, o que não ocorre com as tabelas básicas do Sistema R. Além disso, o QBE fornece suporte explícito aos conceitos de *domínio* e *chave primária* (mas não a chave alternada); é exigido que as chaves primárias sejam únicas e não nulas. O usuário não tem que criar um índice para obter a verificação desta unicidade. No entanto, a Regra de Integridade 2 não é imposta. (Novamente, tal como na SQL, a linguagem completa QBE inclui meios para se especificar constantes arbitrárias de integridade, mas isto não está suportado na implementação corrente.)
- *Ser relacionalmente completa* — A linguagem QBE é completa no sentido dado na Seção 7.2, mas o subconjunto implementado não. (Algumas consultas SQL não podem ser expressas nesse subconjunto.)
- *Extrações do IMS* — O QBE fornece um utilitário para copiar dados de um banco de

dados IMS (veja a parte 3 deste livro) para um banco de dados QBE, tornando-os disponíveis para consulta através do interface normal do QBE.

EXERCÍCIOS

Forneça soluções QBE para os Exercícios 7.1–7.20 e 7.29–7.34.

REFERÊNCIAS E BIBLIOGRAFIA

Algumas abordagens alternativas às linguagens interativas formais que usam terminais de telas estão apresentadas em [11.13] e [11.14]. A referência [11.15] inclui uma avaliação comparativa de um grande número de linguagens de consulta, incluindo QBE e SQL. A referência [11.16] trata de um sistema chamado "System for Business Automation", que é um conjunto ampliado do QBE.

- 11.1 IBM Corporation. "Query-By-Example: Program Description/Operations Manual." IBM Form n° SH20-2077.
- 11.2 IBM Corporation. "Query-by-Example: Terminal User's Guide." IBM Form n° SH20-2078.
- 11.3 M. M. Zloof. "Query By Example." *Proc. NCC 44* (maio de 1975).
- 11.4 M. M. Zloof. "Query By Example: The Invocation and Definition of Tables and Forms." *Proc. 1st International Conference on Very Large Data Bases* (setembro de 1975).
- 11.5 M. M. Zloof. "Query-by-Example: Operations on the Transitive Closure." IBM Research Report RC5526 (julho de 1975).
- 11.6 M. M. Zloof. "Query-by-Example: Operations on Hierarchical Data Bases." *Proc. NCC 45* (1976). Descreve uma versão do QBE baseado em hierarquias ao invés de tabelas. Introduz também uma sintaxe linear.
- 11.7 M. M. Zloof. "Query-by-Example: A Data Base Management Language." IBM Research Report (sendo preparado).
- 11.8 M. M. Zloof. "Query-by-Example: A Data Base Language." *IBM Sys. J.* 16 n° 4 (1977).
- 11.9 M. M. Zloof. "Design Aspects of the Query-by-Example Data Base Management Language." In *Databases: Improving Usability and Responsiveness* (ed., B. Shneiderman). New York: Academic Press (1978).
- 11.10 K. E. Nibuhr and S. E. Smith. "Implementation of Query-by-Example on VM/370." IBM Research Report (sendo preparado).
- 11.11 J. C. Thomas and J. D. Gould. "A Psychological Study of Query-by-Example." *Proc. NCC 44* (maio de 1975).
- 11.12 D. Greenblatt and J. Waxman. "A Study of Three Database Query Languages." In *Databases: Improving Usability and Responsiveness* (ed., B. Shneiderman). New York: Academic Press (1978). Um estudo comparativo do QBE, SQL, e álgebra relacional, do ponto de vista de utilização.
- 11.13 M. E. Senko. "DIAM II with FORAL LP: Making Pointed Queries with Light Pen." *Proc. IFIP Congress* (1977).
- 11.14 N. McDonald and M. R. Stonebraker. "CUPID – The Friendly Query Language." *Proc. ACM Pacific Conference, San Francisco* (abril de 1975). Disponível na ACM Golden Gate Chapter, P. O. Box 24055, Oakland, California 94623. Uma breve introdução à linguagem CUPID, que é implementada sobre QUEL (Veja o Capítulo 13). CUPID é uma linguagem gráfica na qual o usuário pode preparar consultas de complexidade arbitrária, simplesmente manipulando com a caneta seletora um pequeno número de símbolos padrão. Na Fig. 11.3, mostramos uma possível representação CUPID do Exemplo 11.2.8, para ilustrar a rápida compreensão da linguagem.

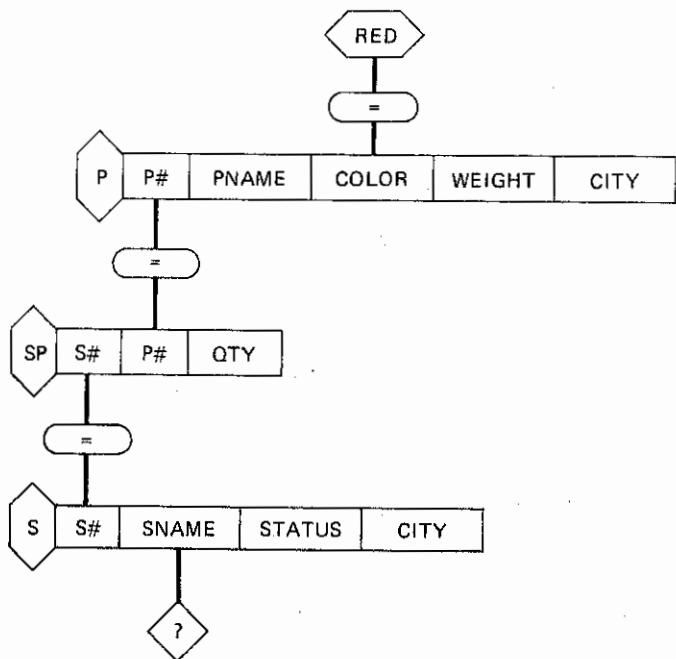


Fig. 11.3 Exemplo de uma consulta em CUPID.

- 11.15 R. N. Cuff. "Database Query Systems for the Casual User." Man-Machine Systems Laboratory, Dept. of Electrical Engineering, University of Essex, Colchester, England CO4 3SQ (março de 1979).

11.16 M. M. Zloof and S. P. de Jong. "The System for Business Automation (SBA): Programming Language." *CACM* 20, nº 6 (junho de 1977)

12

Álgebra Relacional

12.1 INTRODUÇÃO

Discutimos brevemente álgebra relacional na Seção 3.5, como base para uma sublinguagem de dados de alto nível. Naquela seção, chamamos a atenção de que diversas outras linguagens, tais como SQL e QBE, foram desenvolvidas desde que a álgebra foi inicialmente definida, e a maioria dos sistemas relacionais suporta uma dessas linguagens ao invés da álgebra em si. Ainda assim, a álgebra é importante, e neste capítulo nós iremos descrevê-la com algum detalhe.

A álgebra relacional é um conjunto de operações e relações. Cada operação usa uma ou mais relações como seus operandos, e produz outra relação como seu resultado.¹ Codd definiu originalmente um conjunto dessas operações em [12.1], e mostrou que essas operações eram “relacionalmente completas”, no sentido de que possuem pelo menos a potência de recuperação do cálculo relacional (veja Capítulo 13). Foram propostas numerosas variantes da álgebra original desde a publicação de [12.1]; veja, por exemplo, as referências [12.7, 12.12, 12.14, 12.15]. Neste capítulo apresentaremos nossa própria variante — que, entretanto, não difere muito do original — usando-a para introduzir diversos outros conceitos incluindo, em particular, o de *modelo de banco de dados relacional*.

A álgebra que apresentaremos consiste de dois grupos de operadores: o tradicional conjunto de operadores união, interseção, diferença e produto Cartesiano (todos ligeira-

¹ Como o resultado de uma operação de álgebra relacional é uma relação, esta relação pode, por sua vez, ser objeto de novas operações algébricas. Portanto, os operandos de qualquer operação podem ser especificados ou como simples nomes de relações ou como *expressões* que são avaliadas como relações. Em outras palavras, as expressões de álgebra relacional podem estar em ninhos de qualquer profundidade. (Há uma analogia óbvia com as expressões aritméticas em ninhos nas linguagens comuns de programação.) Em termos matemáticos, expressamos o fato de ser o resultado de qualquer operação algébrica uma outra relação dizendo que as relações formam um *sistema fechado* na álgebra. Naturalmente, observações semelhantes aplicam-se ao SQL, QBE e outras linguagens relacionais de alto nível.

mente modificados para operarem sobre relações ao invés de conjuntos arbitrários); e os operadores relacionais especiais seleção, projeção, junção e divisão. Como referência, apresentamos uma sintaxe tipo BNF (Backus-Naur Form) completa da nossa versão da álgebra na Fig. 12.1. Esta sintaxe tem deliberadamente menos palavras do que a da Seção 3.5. Depois discutiremos em detalhes os dois grupos de operadores, com exemplos, nas seções subsequentes. Como sempre, os exemplos estão baseados no banco de dados de fornecedores e peças. Ressaltamos que nossa versão da álgebra inclui regras para nomear os atributos dos resultados, um aspecto que é freqüentemente ignorado.

Nota: Estamos supondo durante todo este capítulo que a ordem dos atributos dentro de uma relação é significativa – não porque isto seja necessário, mas porque simplifica as definições.

Nota: Os colchetes quadrados não são usados na sintaxe a seguir para indicar material opcional, mas sim como símbolos na linguagem sendo definida.

A1	instrução-algébrica	::=	def-de-alias designação
A2	def-de-alias	::=	alias ALIASES nome-de-relação;
A3	designação	::=	nome-de-relação [lista-de-especif-de-atributos-entre-vírgulas] := expr-algébrica ;
A4	lista-de-especif-de-atributos-entre-vírgulas	::=	especif-de-atributos especif-de-atributos, lista-de-especif-de-atributos-entre-vírgulas
A5	especif-de-atributos	::=	nome-de-atributo nome-de-relação . nome-de-atributo alias . nome-de-atributo
A6	expr-algébrica	::=	seleção projeção infixação
A7	seleção	::=	primitiva WHERE expr-bool
A8	primitiva	::=	nome-de-relação alias (expr-algébrica)
A9	projeção	::=	primitiva primitiva [lista-de-especif-de-atributos-entre-vírgulas] projeção oper-infíxacao projeção
A10	infíxacao	=	UNION INTERSECT MINUS TIMES JOIN DIVIDE BY
A11	oper-infíxacao	=
A12	expr-bool	=	especif-de-atributo oper-de-comparação especif-de-valor
A13	comparação	=	

Fig. 12.1 Uma sintaxe para a álgebra relacional.

Algumas observações sobre a sintaxe

1. Está incluída uma operação de designação (A3) para permitir o armazenamento de resultados. As relações mostradas do lado esquerdo de uma designação devem ter sido adequadamente declaradas, como todas as relações com nomes (o que não ocorre com relações derivadas), mas, para maior clareza, os nomes dos seus atributos são mencionados explicitamente na própria designação. Não está mostrada sintaxe para declaração de relação.

2. As categorias *alias*, *nome-de-relação*, e *nome-de-atributo* são todas definidas como sendo *identificadores* (uma categoria terminal com relação a esta sintaxe).

3. A categoria *expr-bool* representa uma expressão Booleana (isto é, uma combinação Booleana de *comparações*, formada de acordo com as regras normais). A categoria *comparação* representa uma comparação singela entre um valor de atributo (representado por uma *especif-de-atributo*, isto é, um nome de atributo qualificado ou não) e uma constante ou outro valor de atributo (representado por *especif-de-valor*).

12.2 OPERAÇÕES TRADICIONAIS COM CONJUNTOS

As operações tradicionais com conjuntos são união, interseção, diferença, e produto Cartesiano. Em todas, com exceção do produto Cartesiano, as duas relações operando têm que ser *compatíveis de união*, isto é: elas têm que ser do mesmo grau, digamos n , e os j -ésimos atributos das duas relações (j na faixa de 1 a n) têm que ser retirados do mesmo domínio (não precisam ter o mesmo nome).

União

A união de duas relações (compatíveis de união) A e B , $A \text{ UNION } B$, é o conjunto de todas as tuplas t pertencentes a A ou B (ou ambos).

Exemplo. Seja A o conjunto de tuplas dos fornecedores de Londres, e B o conjunto de tuplas dos fornecedores que fornecem a peça P1. Então $A \text{ UNION } B$ é o conjunto de tuplas de fornecedores que ou estão localizados em Londres ou fornecem a peça P1 (ou ambos).

Interseção

A interseção de duas relações (compatíveis de união) A e B , a $\text{INTERSECT } B$, é o conjunto de todas as tuplas t pertencentes a A e B .

Exemplo. Sejam A e B como no exemplo de “União” acima. Então $A \text{ INTERSECT } B$ é o conjunto de tuplas de fornecedores que estão localizados em Londres e fornecem a peça P1.

Diferença

A diferença entre duas relações (compatíveis de união) A e B (nessa ordem), a $\text{MINUS } B$, é o conjunto de todas as tuplas t pertencentes a A mas não a B .

Exemplo. Sejam novamente A e B como no exemplo “União”. Então $A \text{ MINUS } B$ é o conjunto de tuplas de fornecedores que estão localizados em Londres e *não* fornecem a peça P1. (O que é $B \text{ MINUS } A$?)

Produto Cartesiano Estendido

O produto Cartesiano estendido de duas relações A e B , $A \text{ TIMES } B$, é o conjunto de todas as tuplas t tais que t é a concatenação de uma tupla a pertencente a A com uma tupla b pertencente a B . A concatenação de uma tupla $a = (a_1, \dots, a_m)$ com uma tupla $b = (b_{m+1}, \dots, b_{m+n})$ – nessa ordem – é a tupla $t = (a_1, \dots, a_m, b_{m+1}, \dots, b_{m+n})$.

Exemplo. Seja A o conjunto de todos os números de fornecedores, e B o conjunto de todos os números de peças. Então $A \text{ TIMES } B$ é o conjunto de todos os possíveis pares número de fornecedor/número de peça.

É fácil verificar-se que UNION é associativo – isto é, se X , Y e Z são *projeções* arbitrárias (no sentido da Fig. 12.1), então as expressões $(X \text{ UNION } Y) \text{ UNION } Z$ e $X \text{ UNION }$

(Y UNION Z) são equivalentes. Por conveniência, entretanto, escreveremos a sequência de UNIONs sem parêntesis embutidos; por exemplo, cada uma das expressões anteriores pode ser simplificada sem ambigüidades para X UNION Y UNION Z. A mesma observação se aplica a INTERSECT e TIMES, mas não a MINUS.

12.3 NOMES-ATRIBUTO PARA RELAÇÕES DERIVADAS

Embora estejamos considerando os atributos como estando ordenados da esquerda para a direita dentro de uma relação, ainda nos baseamos nos nomes de atributos para fins de referências, e ainda exigimos que nenhuma relação tenha dois atributos com o mesmo nome. Estamos supondo que esta restrição é automaticamente forçada para as relações declaradas ou "básicas". Nesta seção introduziremos uma regra para a geração de nomes de atributos para relações *derivadas* — isto é, relações representadas por expressões ao invés de por nome ou alias (pseudônimo) (veja abaixo) — de acordo com a restrição de unicidade de nome.

Primeiro, seja R o nome de uma relação declarada, e seja A o nome de um atributo dentro de R. A é dito ser um nome *não qualificado* para este atributo. Em uma relação declarada, dois atributos não podem ter o mesmo nome não qualificado. Seja S um pseudônimo de R, introduzido pela definição de alias

S ALIAS R;

(S é simplesmente um outro nome da relação R). Quando a relação dada é referenciada pelo nome R, o atributo A é considerado como tendo o nome qualificado R.A. Quando a relação dada é referenciada pelo pseudônimo S, o atributo A é considerado como tendo um nome *qualificado* S.A. Portanto, um atributo de uma relação declarada sempre tem um nome qualificado, pois um atributo só pode ser referenciado no contexto da relação que o contém, e a relação que o contém tem que ser referenciada ou por um nome ou por um pseudônimo; mas o atributo pode ser referenciado por seu nome não qualificado, se isto não causar ambigüidades.

Os atributos de relações derivadas também têm sempre nomes qualificados, gerados como descrito a seguir. (Novamente, eles poderão ser referenciados por seus nomes não qualificados, caso isto não cause ambigüidade.) Como já dito anteriormente, relação derivada é aquela representada por uma expressão, ao invés de por seu nome ou pseudônimo. Ao considerar estas derivações, nós podemos obviamente restringir nossa atenção a expressões que envolvem exatamente um dos operadores algébricos (pois os operandos daquele operador podem ser, por sua vez, relações derivadas).

União, Interseção, Diferença

O resultado tem os mesmos nomes de atributos qualificados do primeiro operando.

Produto Cartesiano estendido

Consideremos o produto A TIMES B. Sejam os nomes de atributos qualificados de A e B, na ordem da esquerda para a direita,

$$A.A_1, \dots, A.A_m \quad \text{and} \quad B.B_{(m+1)}, \dots, B.B_{(m+n)},$$

respectivamente. Então os atributos de A TIMES B terão exatamente esses nomes de atributos qualificados (na ordem da esquerda para a direita).

Exemplo. Sejam as três relações A(S#), B(P#), e D(S#, P#). A TIMES B tem os atributos

(A.S#, B.P#). Se chamarmos a este produto de C, então C TIMES D tem como atributos (A.S#, B.P#, D.S#, D.P#).

Como o exemplo mostra, a relação derivada pode ter nomes não qualificados que não são únicos, mas seus nomes qualificados têm que ser únicos. Suponhamos que precisamos formar o produto Cartesiano estendido de uma relação R com ela mesma. A expressão R TIMES R é ilegal, pois viola a regra de unicidade de nomes. Portanto, temos que introduzir um pseudônimo, digamos R₁:

R1 ALIASES R;

Agora podemos escrever R₁ TIMES R, ou R TIMES R₁, gerando o produto requerido sem violar a regra de unicidade de nome.

Operações relacionais especiais

Veja a seção seguinte.

12.4 OPERAÇÕES RELACIONAIS ESPECIAIS

Seleção

O operador algébrico de seleção (não confundir com o SELECT da SQL) produz um subconjunto “horizontal” de uma dada relação – isto é, aquele subconjunto de tuplas dentro da relação dada que satisfaz a um predicado especificado. O predicado é expresso como uma combinação Booleana de termos, sendo cada termo uma comparação singela que pode se tornar verdadeira ou falsa para uma determinada tupla, inspecionando-se aquela tupla isoladamente. (Se um termo envolver uma comparação entre valores de dois atributos dentro da tupla, então esses atributos têm que estar definidos no mesmo domínio.) Se X denotar a relação sobre a qual se está fazendo a seleção, então o resultado da seleção terá exatamente os mesmos nomes de atributos qualificados que X (note que X pode ser uma expressão).

Na Fig. 12.2 estão mostrados alguns exemplos de seleção.

S WHERE CITY = 'LONDON'

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

P WHERE WEIGHT < 14

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P5	Cam	Blue	12	Paris

SP WHERE S# = 'S1'
AND P# = 'P1'

S#	P#	QTY
S1	P1	300

Fig. 12.2 Três exemplos de seleção.

A sintaxe da Fig. 12.1 permite que a “relação dada” de uma seleção seja especificada por nome (ou pseudônimo), ou colocada dentro de parêntesis, como qualquer expressão da álgebra relacional. Daremos mais exemplos na seção 12.5.

Projeção

O operador projeção produz um subconjunto “vertical” de uma dada relação – isto é, o subconjunto obtido pela seleção de atributos especificados, em uma ordem especificada da esquerda para a direita, e depois eliminando as tuplas duplicadas dentro dos atributos selecionados. Como estamos atribuindo significado à ordem dos atributos dentro de uma relação, a projeção nos fornece a maneira de permutar (reordenar) os atributos de uma dada relação. Se X denotar a relação a ser projetada, então o resultado da projeção terá os mesmos nomes de atributos qualificados que X (novamente, X pode ser uma expressão). Nenhum atributo pode ser especificado mais de uma vez em uma operação de projeção. A omissão da lista de nomes de atributos é equivalente à especificação de uma lista contendo todos os nomes de atributos da relação dada, na sua ordem correta, da esquerda para a direita (em outras palavras, tal projeção é idêntica à relação dada).

A Fig. 12.3 mostra alguns exemplos de projeção.

A sintaxe da Fig. 12.1 permite que a “relação dada” de uma projeção seja especificada por um nome ou pseudônimo, ou colocada entre parêntesis como qualquer expressão relacional. Novamente, daremos mais exemplos na seção 12.5.

Junção

Na Seção 3.5 nós introduzimos o *equijoin*, e mencionamos que este é apenas um dos muitos operadores possíveis de junção. Como um exemplo, nós podemos definir a junção *maior do que* da relação A sobre o atributo X com a relação B sobre o atributo Y como o conjunto de todas as tuplas *t* tais que *t* seja a concatenação de uma tupla *a* pertencente a A e uma tupla *b* pertencente a B, onde *x > y* (sendo *x* o componente X de A e *y* o componente Y de B). Deve ficar claro, entretanto, que essas junções não são essenciais; por definição, são equivalentes a se fazer o produto Cartesiano estendido das duas relações dadas seguido de uma seleção adequada naquele produto. A junção maior do que acima, aliás, produz o mesmo resultado que a expressão

(A TIMES B) WHERE A.X > B.Y

Há uma junção específica, entretanto, que é tão comum que se torna útil uma observação explícita sobre ela. É a junção *natural*. Lembre-se da Seção 3.5 que a junção natural é

S [CITY]
CITY
London
Paris
Athens

S [SNAME, CITY, S#, STATUS]			
SNAME	CITY	S#	STATUS
Smith	London	S1	20
Jones	Paris	S2	10
Blake	Paris	S3	30
Clark	London	S4	20
Adams	Athens	S5	30

Fig. 12.3 Dois exemplos de projeção.

uma equijoin em que foram eliminadas as colunas duplicadas. A sintaxe da Fig. 12.1 fornece um importante caso especial de junção natural, no qual os atributos comuns das duas relações têm os mesmos nomes não qualificados. A expressão A JOIN B é definida se e somente se, para cada nome de atributo não qualificado que seja comum a A e B, o domínio básico seja o mesmo para ambas as relações. Suponhamos que esta condição está satisfeita. Sejam os nomes qualificados dos atributos de A e B, na sua ordem da esquerda para a direita,

$$A.A_1, \dots, A.A_m \quad \text{and} \quad B.B_{(m+1)}, \dots, B.B_{(m+n)},$$

respectivamente; sejam C_i, \dots, C_j os nomes não qualificados dos atributos que são comuns a A e B; e sejam B_r, \dots, B_s os nomes não qualificados dos atributos restantes de B (sem alteração de sua ordem relativa) após a remoção de C_i, \dots, C_j . Então A JOIN B é definida como sendo equivalente a

```
(A TIMES B)[A.A1, . . . , A.Am, B.Br, . . . , B.Bs]
WHERE A.Ci=B.Ci
      AND . . . . .
      AND A.Cj=B.Cj
```

Daqui em diante usaremos “join” para significar esta junção natural, a menos de especificação em contrário. Por conveniência, escreveremos as sequências de JOINs sem parêntesis embutidos; por exemplo, as expressões (X JOIN Y) JOIN Z e X JOIN (Y JOIN Z) podem ser ambas simplificadas sem causar ambigüidade para X JOIN Y JOIN Z, pois JOIN é associativo. (*Exercício*: prove a última afirmativa.) Ressaltamos que, se A e B não tiverem nomes comuns de atributos, então A JOIN B é idêntico a A TIMES B. A Fig. 12.4 mostra a junção natural das relações S e SP (incluindo os nomes qualificados gerados para o resultado).

S.S#	S.SNAME	S.STATUS	S.CITY	SP.P#	SP.QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
S1	Smith	20	London	P3	400
S1	Smith	20	London	P4	200
S1	Smith	20	London	P5	100
S1	Smith	20	London	P6	100
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
S3	Blake	30	Paris	P2	200
S4	Clark	20	London	P2	200
S4	Clark	20	London	P4	300
S4	Clark	20	London	P5	400

Fig. 12.4 Junção de S e SP em S# (S JOIN SP).

DEND	S#	P#
	S1	P1
	S1	P2
	S1	P3
	S1	P4
	S1	P5
	S1	P6

S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

DOR	P#
	P1
DOR	P#
	P2
	P4
DOR	P#
	P1
	P2
	P3
	P4
	P5
	P6

DEND DIVIDE BY DOR		
S#		
S1		
S2	S#	
	S1	
	S4	
		S#
		S1

Fig. 12.5 Três exemplos de divisão.

Divisão

O operador divisão divide uma relação dividendo A de grau $m + n$ por uma relação divisor B de grau n , e produz uma relação resultado de grau m . O atributo $(m + i)$ -ésimo de A e o i -ésimo atributo de B (i na faixa de 1 a n) têm que ser definidos no mesmo domínio. Consideremos os m primeiros atributos de A como um atributo composto simples X, e o último n como outro, Y; então A pode ser visto como um conjunto de pares de valores $\langle x, y \rangle$. De forma semelhante, B pode ser visto como um conjunto de valores singelos, $\langle y \rangle$. Então, o resultado da divisão de A por B — isto é, A DIVIDED BY B — é o conjunto de valores x tais que o par $\langle x, y \rangle$ aparecem em A para todos os valores y que aparecem em B. Os atributos do resultado têm os mesmos nomes qualificados que os primeiros m atributos de A.

A Fig. 12.5 mostra alguns exemplos de divisão. O dividendo em cada caso (DEND) é a projeção de SP em (S#, P#); os divisores (DOR) estão indicados na figura.

12.5 EXEMPLOS

12.5.1 Obtenha os nomes dos fornecedores que fornecem a peça P2. (Exemplo 7.2.7)

Mostraremos primeiro uma solução por etapas:

```
TEMP1 [S#, SNAME, STATUS, CITY, P#, QTY]
       := S JOIN SP ;
TEMP2 [S#, SNAME, STATUS, CITY, P#, QTY]
       := TEMP1 WHERE P# = 'P2' ;
RESULT [SNAME]   := TEMP2 [SNAME] ;
```

Usando uma expressão em ninho:

```
((S JOIN SP) * WHERE P# = 'P2') [SNAME]
```

O resultado desta expressão tem um atributo, com o nome qualificado S.SNAME.

12.5.2 Obtenha os números dos fornecedores que fornecem pelo menos uma peça vermelha. (Exemplo 7.2.10)

```
((P WHERE COLOR = 'RED') [P#] JOIN SP) [S#]
```

Nome resultante do atributo: SP.S#.

12.5.3 Obtenha os nomes dos fornecedores que fornecem todas as peças. (Exemplo 7.2.18)

```
((SP[S#, P#] DIVIDE BY P[P#]) JOIN S) [SNAME]
```

Nome resultante do atributo: S.SNAME.

12.5.4 Obtenha os números dos fornecedores que fornecem pelo menos todas as peças fornecidas pelo fornecedor S2. (Exemplo 7.2.19)

```
SP[S#, P#] DIVIDE BY (SP WHERE S# = 'S2') [P#]
```

Nome resultante do atributo: SP.S#

12.5.5 Obtenha os nomes dos fornecedores que não fornecem a peça P2. (Exemplo 7.2.14)

```
((S[S#] MINUS (SP WHERE P# = 'P2')) [S#]) JOIN S) [SNAME]
```

Nome resultante do atributo: S.SNAME.

12.5.6 Obtenha todos os pares de números de fornecedores de forma que os dois fornecedores estejam localizados na mesma cidade. (Exemplo 7.2.6)

```
FIRST ALIASES S;
SECOND ALIASES S;
((FIRST TIMES SECOND)
    WHERE FIRST.CITY = SECOND.CITY
        AND FIRST.S# < SECOND.S# )
    [FIRST.S#, SECOND.S#]
```

Nomes resultantes dos atributos: FIRST.S#, SECOND.S#

Concluiremos esta seção com uma nota sobre operações de atualização. A álgebra relacional é basicamente uma linguagem de recuperação. UNION e MINUS *poderiam* ser usados para operações de inserção e remoção; por exemplo,

```
P := P UNION {<'P7','WASHER','GREY',2,'ATHENS'>} ;
SP := SP MINUS {<'S1',?,?>} ;
```

(A primeira destas designações insere a tupla de P7 na relação P; a segunda remove todas as tuplas de S1 da relação SP.) Entretanto, UNION e MINUS não são substitutos satisfatórios de INSERT e DELETE, pois não manuseiam adequadamente as situações de erro (por “situações de erro” queremos significar condições que são costumeiramente tratadas como erro por INSERT e DELETE). UNION, por exemplo, não rejeitará uma tentativa de “inserção” de uma tupla que seja duplicata de uma já existente; e MINUS não rejeitará uma tentativa de “remoção” de uma tupla inexistente. Na prática, portanto, um sistema que suporte uma DML baseada na álgebra relacional deve prover INSERT e DELETE explícitos (e também UPDATE) como operadores.

12.6 DISCUSSÃO

Dissemos no início deste capítulo que a álgebra ainda é importante, embora seja menos “amiga do usuário” do que linguagens como SQL e QBE. A razão básica de sua importância é que ela fornece uma medida para comparação das outras linguagens. Uma vez que a álgebra é relationalmente completa, para se mostrar que uma outra linguagem L é também completa nesse sentido basta mostrar que L inclui operadores equivalentes aos da álgebra. (Veja Exercício 12.2.) A álgebra também estabelece as fundações para pesquisa sobre diversos outros aspectos da administração de um banco de dados, tais como projeto de bancos de dados, definição de visões, e reestruturação. Nós já abordamos alguns desses tópicos no Capítulo 9 (nesse capítulo falávamos em termos de SQL ao invés de álgebra, mas a maioria dos operadores SQL possui equivalentes diretos na álgebra); e o Capítulo 14 irá mostrar o papel fundamental desempenhado pela projeção e pela junção natural na área de projeto de bancos de dados. As referências [12.8–12.11] também usam a álgebra como base para um importante trabalho de técnicas de otimização. (Veja também o livro de Ullman [13.6], que inclui um capítulo de levantamentos sobre otimização de consultas.)

Estamos agora prontos para definir o *modelo relacional de banco de dados*.² O modelo relacional consiste de dois componentes principais:

- A estrutura relacional de dados definida no Capítulo 4; e
- a álgebra relacional.

Um sistema de banco de dados pode ser chamado de *completamente relacional* [4.6] se suportar

- bancos de dados relacionais (incluindo os conceitos de domínio e chave e as duas regras de integridade); e
- uma linguagem que seja pelo menos tão potente quanto a álgebra relacional (e que

² Nosso uso deste termo difere em relação às edições anteriores deste livro.

assim permaneça mesmo que todos os dispositivos de *loops* e recursões tenham que ser removidos).³

Um sistema que suporte bancos de dados relacionais mas cuja linguagem seja menos potente do que a álgebra pode ser chamado de *semi-relacional* [4.6].

De acordo com essas definições, observamos que provavelmente não existem hoje (1981) sistemas que sejam completamente relacionais, ou até mesmo semi-relacionais! As definições relacionais mudaram um pouco nos últimos dez anos; as críticas de que o "sistema relacional" é um objetivo móvel não são infundadas. Mas seria mais preciso dizer-se que os conceitos *evoluíram*, não se modificaram de forma ampla ou imprevisível. Poucos, se é que algum, dos princípios originais foram superados. Por exemplo, a definição inicial do modelo de banco de dados relacional é ainda válida hoje basicamente; porém pesquisas mais recentes resultaram em várias adições e clarificações à definição original (as duas regras de integridade estão nesse caso).

Podemos também predizer que certamente os conceitos continuarão a evoluir. Uma área que correntemente vem recebendo muita atenção é a dos valores nulos. Diversos investigadores já propuseram melhorias à álgebra para lidar com nulos [12.17, 12.18, 4.6]. A referência [4.6], por exemplo, sugere (entre outras coisas) duas variantes do operador junção: a "junção pode ser", na qual as tuplas são unidas não na base de alguma condição ser *verdadeira*, mas na base de ser *desconhecida* (nula); e a "junção externa", na qual as tuplas de uma relação sem correspondentes na outra aparecem no resultado concatenadas com uma tupla toda nula. Entretanto, tais propostas devem ser olhadas como preliminares neste momento.

EXERCÍCIOS

12.1 Nós já ressaltamos que a junção não é uma operação essencial, podendo ser definida em termos de operadores mais primitivos. O mesmo é verdadeiro para interseção e divisão (portanto as verdadeiras primitivas são união, diferença, produto, seleção e projeção). Defina interseção e divisão em termos dessas cinco primitivas.

12.2 Mostre que a SQL é relationalmente completa, mostrando que ela inclui análogas das cinco primitivas algébricas.

12.3 Dado o banco de dados da Fig. 4.7, avalie a expressão

S JOIN SP JOIN P

(Cuidado: há uma armadilha aqui.)

12.4 Dê soluções algébricas aos Exercícios 7.1–7.28.

REFERÊNCIAS E BIBLIOGRAFIA

As referências [12.1–12.18] são especificamente voltadas para a álgebra relacional. As outras referências descrevem linguagens e sistemas que têm pouca ligação com a álgebra em si, mas são de qualquer forma relevantes.

12.1 E. F. Codd. "Relational Completeness of Data Base Sublanguages." In *Data Base Systems*, Courant Computer Science Symposia Series, Vol. 6. Englewood Cliffs, N. J.: Prentice-Hall (1972).

Este artigo inclui uma definição formal do cálculo relacional (veja o capítulo 13), e introduz o conceito de ser relationalmente completo. Uma linguagem é dita ser relationalmente completa

3

Esta condição é equivalente a se insistir em que a linguagem seja relationalmente completa no sentido mais exigente do termo (veja as observações no final da Seção 7.2).

se possuir a propriedade de que qualquer relação definível por meio de expressões de cálculo possa ser recuperada por instruções adequadas daquela linguagem. (Como já mencionado, a nossa definição de ser relacionalmente completa da Seção 7.2 é mais exigente do que esta: ela estipula que qualquer relação definível por uma *única* expressão de cálculo possa ser recuperada por uma *única* instrução da linguagem.)

Este artigo também fornece uma definição formal da álgebra relacional, e prova que esta álgebra é completa fornecendo um algoritmo ("algoritmo de redução de Codd") para converter uma expressão arbitrária de cálculo em uma expressão algébrica semanticamente equivalente (assim demonstrando uma possível abordagem para implementação do cálculo). O artigo conclui com uma breve seção comparando e contrastando o cálculo e a álgebra como candidatos a sublinguagens de dados.

12.2 R. C. Goldstein and A. J. Strnad. "The MacAIMS Data Management System." *Proc. 1970 ACM SIGFIDET Workshop on Data Description and Access.*

12.3 A. J. Strnad. "The Relational Approach to the Management of Data Bases." *Proc. IFIP Congress 1971.*

O MacAIMS parece ter sido o mais antigo exemplo de um sistema suportando uma linguagem de dados tanto para relações n -árias como com orientação para conjuntos. A linguagem é algébrica. Dois dispositivos interessantes do MacAIMS são os seguintes.

- A estrutura de armazenamento pode variar de relação para relação (permitindo assim que cada relação seja armazenada da forma que lhe for mais adequada). Para cada estrutura, um "módulo de estratégia relacional" mantém as relações apropriadamente e permite que o usuário as veja na forma tabular.
- Os atributos são armazenados como "conjuntos de elementos de dados". Cada elemento de dado (valor de atributo) recebe um único número de referência de comprimento fixo, e todas as referências ao elemento de dado dentro da relação são feitas via o número de referência. O algoritmo para designação do número de referência é tal que, se A e B pertencerem ao mesmo conjunto de elementos de dados, o número de referência de A será maior do que o de B se e somente se A for maior do que B. Como resultado, qualquer operação de comparação entre dois elementos de dados (do mesmo conjunto de elementos de dados) pode ser feita diretamente sobre os correspondentes números de referência; além disso, a própria comparação será provavelmente mais eficiente, pois os números de referência são de comprimento fixo, enquanto que os elementos de dados podem ser de comprimento variável. Isto é particularmente significativo tendo em vista o fato de que essas comparações são as operações mais freqüentemente executadas no sistema.

O sistema RDMS [12.4], que está atualmente em uso nos departamentos administrativos do M.I.T., é aparentemente uma forma atualizada do MacAIMS.

12.4 J. Stewart and J. Goldman. "The Relational Data Management System: A Perspective." *Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control.*

12.5 S. J. P. Todd. "The Peterlee Relational Test Vehicle – A System Overview." *IBM Sys. J.* 15, nº 4 (1976).

PRTV é um sistema experimental desenvolvido pela IBM UK Scientific Centre em Peterlee, Inglaterra. Ele é baseado em um protótipo anterior chamado IS/1 [12.6]. Suporta relações n -árias e uma versão da álgebra baseada em propostas documentadas em [12.7]. Três aspectos significativos do PRTV são.

1. Incorpora algumas técnicas interessantes de otimização de alto nível [12.9].
2. Inclui um dispositivo de "avaliação retardada", que é importante tanto para a otimização [12.9] como para a provisão de visões. Isto é, uma instrução relacional de designação pode ser tratada como uma definição de visão, ao invés de ser executada imediatamente.
3. Provê "extensibilidade de funções" – isto é, a possibilidade de que o usuário estenda o sistema para incluir um conjunto arbitrário de funções integradas.

12.6 M. G. Notley. "The Peterlee IS/1 System." IBM (UK) Scientific Centre Report UKSC-0018 (março de 1972), IBM (UK) Scientific Centre, Neville Rd., Peterlee, Co. Durham, Inglaterra HA2 7HH.

12.7 P. A. V. Hall, P. Hitchcock, and S. J. P. Todd. "An Algebra of Relations for Machine Computation." *Conference Record of the Second ACM Symposium on Principles of Programming Languages* (1975).

12.8 J. M. Smith and P. Y. T. Chang. "Optimizing the Performance of a Relational Database Interface." *CACM* 18, nº 10 (outubro de 1975).

Um artigo extremamente claro explicando os algoritmos usados no "Smart Query Interface for a Relational Algebra" (SQUIRAL). As técnicas de otimização empregadas incluem:

- Transformação da expressão algébrica original em uma seqüência de operações equivalente porém mais eficiente.
- Designação de operações distintas sobre a expressão transformada como tarefas distintas e exploração da concorrência e troca de informações entre elas.
- Coordenação das ordens de seqüenciamento das relações temporárias passada entre essas tarefas.

O SQUIRAL também procura explorar quaisquer índices que possam existir e localizar referências de páginas armazenadas.

12.9 P. A. V. Hall. "optimisation of a Single Relational Expression in a Relational Data Base System." *IBM J. R & D.* 20, nº 3 (1976).

Este artigo descreve algumas das técnicas de otimização usadas no sistema PRTV [12.5]. O PRTV, como o SQUIRAL [12.8], procura transformar uma dada expressão algébrica em uma forma mais eficiente antes de avaliá-la. Uma característica do PRTV é que o sistema não avalia automaticamente cada expressão assim que a recebe; combina cada nova expressão com as que já recebeu, construindo uma expressão maior e mais complexa, retardando a avaliação para o último momento possível. Assim, o "Single relational expression" do título do artigo pode representar uma seqüência completa de operações do usuário. As otimizações descritas parecem-se com as do SQUIRAL, mas vão além em alguns aspectos; resumidamente são como se segue (por ordem de aplicação).

- Move as operações SELECT para que elas sejam executadas o mais cedo possível.
- Combina seqüências de operações PROJECT.
- Elimina operações redundantes; simplifica expressões envolvendo relações vazias e predicados triviais.
- Elimina subexpressões comuns.

O artigo conclui com alguns resultados experimentais e algumas sugestões para investigações adicionais.

12.10 F. P. Palermo. "A Data Base Search Problem." *Information Systems: COINS IV* (ed., J. T. Tou). New York: Plenum Press (1974).

Este artigo apresenta um método de implementação para uma instrução de recuperação arbitrária do cálculo relacional. O método é baseado no algoritmo de redução de Codd [12.1], mas introduz várias técnicas interessantes de otimização. Especificamente, são feitas as seguintes melhorias (entre outras) ao algoritmo básico. (Supondo-se que a tupla seja a unidade de acesso ao banco de dados armazenado.)

- Nenhuma tupla é jamais recuperada mais de uma vez.
- Valores desnecessários são descartados de uma tupla assim que ela é recuperada ("valores desnecessários" podendo ser valores correspondentes a atributos não referenciados na consulta ou valores usados somente para fins de seleção – por exemplo, valores de SP.P# no Exemplo 12.5.1). Isto é equivalente a se projetar a relação nos atributos envolvidos, reduzindo consequentemente não só o espaço requerido para cada tupla, mas também a quantidade de tuplas a serem retidas.
- O método usado na construção da relação resultado é o do "princípio do menor crescimento", de forma que o resultado tende a crescer lentamente. Isto traz como efeito a redução tanto do número de comparações envolvidas como da quantidade de memória intermediária necessária.

■ É empregada uma técnica eficiente na construção de junções, compondo dinamicamente os valores usados em “termos de junção” (tais como $S.S\# = SP.S\#$) em “semijunções” (que, efectivamente, são um tipo de índice secundário), e usando uma representação interna de cada junção chamada “junção indireta” (que usa os números de referência internos das tuplas para representá-las). Essas técnicas foram projetadas para reduzir a busca necessária para a construção de uma junção, garantindo que para cada termo de junção as tuplas envolvidas estejam (logicamente) ordenadas pelos valores de seus atributos relevantes. Elas também permitem a determinação dinâmica da “melhor” seqüência a ser usada no acesso às relações requeridas do banco de dados.

12.11 M. W. Blasgen and K. P. Eswaran. “On the Evaluation of Queries in a Relational Data Base System.” IBM Research Report RJ 1745 (abril de 1976).

Diversas técnicas para manipulação de consultas envolvendo operações de projeção, junção e seleção, comparadas na base de seus custos em acessos à memória secundária. As conclusões são de que a aglomeração física de itens logicamente relacionados é um parâmetro crítico de desempenho, e que, na ausência dessa aglomeração, os métodos que parecem mais satisfatórios são os que dependem de ordenação.

12.12 R. M. Pecher. “Efficient Evaluation of Expressions in a Relational Algebra.” *Proc. ACM Pacific Conference, San Francisco* (abril de 1975). Disponível na ACM Golden Gate Chapter P. O. Box 24055, Oakland, California 94623.

Este artigo começa introduzindo uma versão algo modificada da álgebra de [12.1]. As revisões foram motivadas por considerações de eficiência. É discutida a implementação de operadores individuais desta álgebra; supõe-se que as relações sejam armazenadas como tabelas ordenadas, só podendo receber acesso de acordo com sua seqüência de armazenamento. São dados limites de desempenho para cada operador. De acordo com as suposições feitas, os operadores que exigem atenção mais apurada são projeção e divisão. Para esses dois operadores, a conclusão é de que a melhor abordagem é a da ordenação dos dados antes da operação; o artigo mostra que, para uma extensa classe de expressões algébricas, podem ser obtidos resultados intermediários na ordem desejada sem custos extras. O artigo também considera a transformação de expressões em uma forma equivalente mais eficiente, usando algumas técnicas de Palermo [12.10].

12.13 L. R. Gotlieb. “Computing Joins of Relations.” *Proc. 1975 ACM SIGMOD International Conference on the Management of Data*.

Apresenta e compara alguns algoritmos para implementação da junção natural.

12.14 A. L. Furtado and L. Kerschberg. “An Algebra of Quotient Relations.” *Proc. 1977 ACM SIGMOD International Conference on Management of Data* (agosto de 1977).

Apresenta uma álgebra relacional revisada para operação direta sobre “relações quociente”. Dada uma relação n -ária R , pode ser derivada uma relação quociente correspondente de R agrupando-se as tuplas na base dos valores de algum atributo de R (veja a descrição de GROUP BY no Capítulo 7). Por exemplo, a relação quociente derivada da relação SUPPLIER (Fig. 4.6) na base dos valores de CITY é um conjunto de três grupos de tuplas: um contendo duas tuplas de Londres, um contendo duas tuplas de Paris, e um contendo uma única tupla de Atenas. Os autores declararam que a operação direta sobre relações quociente leva à formulação mais natural de consultas e a um potencial de implementação mais eficiente.

12.15 T. H. Merrett. “The Extended Relational Algebra, A Basis for Query Languages.” In *Databases: Improving Usability and Responsiveness* (ed., B. Shneiderman). New York: Academic Press (1978).

Propõe a introdução de quantificadores na álgebra – não apenas os quantificadores existenciais e universais do cálculo (Capítulo 13), mas os quantificadores mais gerais “a quantidade de” e “a proporção de”. Esses quantificadores permitem a expressão de condições tais como “pelo menos três de”, “não mais da metade de”, “Uma quantidade ímpar de”.

12.16 A. V. Aho, C. Beeri, and J. D. Ullman. “The Theory of Joins in Relational Databases.” *ACM Transactions on Database Systems* 4, nº 2 (setembro de 1979).

A definição de junção natural dada na Seção 12.3 é baseada em uma dada neste artigo.

12.17 I. J. Heath. Private communication (abril de 1971).

12.18 M. Lacroix and A. Pirotte. "Generalized Joins." *SIGMOD Record* (bulletin of ACM SIGMOD) 8, nº 3 (setembro de 1976).

12.19 CODASYL Development Committee. "An Information Algebra." *CACM* 5 nº 4 (abril de 1962).
A álgebra de informação é um precursor interessante da álgebra relacional.

12.20 D. L. Childs. "Description of a Set-Theoretic Data Structure." *Proc. FJCC* 33 (1968).

STDs é um sistema no qual é dada ênfase em relações gerais ao invés de relações como tais. A linguagem STDs inclui o conjunto tradicional de operações (união, interseção etc.), mas não operações relacionais como junção.

12.21 J. A. Feldman and P. D. Rovner. "An Algol-Based Associative Language." *CACM* 12, nº 8 (agosto de 1969).

Este artigo descreve a linguagem LEAP e sua implementação. LEAP é essencialmente uma extensão do Algol 60 que possibilita operações de manipulação de conjuntos (não só união, interseção etc., mas também uma poderosa operação de *loop*, que permite ao usuário manipular os conjuntos individuais de elementos um por um). Os elementos de conjunto podem ser de itens singelos ou "associações", isto é, triplas da forma < atributo, objeto, valor >. Um conjunto de "associações" corresponde a uma relação binária na qual "atributo" é o nome da relação (todas as triplas nesse conjunto têm o mesmo "atributo" componente) e, dentro de cada tripla, "objeto" e "valor" são os dois itens associados. Os conjuntos de associações são implementados via um esquema complexo de randomização que, por meio de redundância de dados, permite que qualquer "operação associativa" seja manuseada de uma maneira razoavelmente eficiente.

12.22 W. Ash and E. H. Sibley. "TRAMP: An Interpretive Associative Processor with Deductive Capabilities." *Proc. ACM 23rd Nat. Conf.* (1968).

TRAMP é um sistema que, como o LEAP [12.21], foi projetado para processamento associativo. Como o LEAP, trabalha em termos de < atributo, objeto, valor > como triplas, isto é: armazena relações binárias. A estrutura de armazenamento é novamente baseada em randomização; entretanto, não há redundância de dados – ao invés, é usado um complexo sistema de indicadores de localização para dar a flexibilidade desejada. A diferença realmente significativa entre o TRAMP e o LEAP, no entanto, é que o TRAMP permite ao usuário estabelecer a definição de uma relação (binária ou unária) em termos de relações armazenadas (binárias), usando os operadores "inversão" e "composição"; por exemplo, a relação "pai de" pode ser definida como o inverso de "filho de". Solicitações em termos de uma relação assim definida são então dinamicamente interpretadas em termos das relações realmente armazenadas.

12.23 R. E. Levein and M. E. Maron. "A Computer System for Inference Execution and Data Retrieval." *CACM* 10, nº 11 (novembro de 1967).

Este artigo descreve o "arquivo de dados relacional" e uma linguagem para recuperar dados desse. O arquivo de dados relacional é essencialmente uma coleção de relações binárias, armazenadas com alto grau de redundância de dados para possibilitar resposta eficiente nas operações de recuperação. Como o TRAMP [12.22], o usuário pode definir regras para derivar relações das que estão armazenadas.

12.24 P. J. Titman. "An Experimental Data Base System Using Binary Relations." *Proc. IFIP TC-2 Working Conference on Data Base Management Systems* (eds., Klimbie and Koffeman) (abril de 1974). North-Holland, 1974.

O objetivo principal do protótipo descrito neste artigo foi o de avaliar uma estrutura específica de armazenamento e uma estratégia de acesso associada; por isso foi dada ênfase aos aspectos do sistema de banco de dados que se encontram abaixo do interface do usuário. As "relações binárias" do título seriam mais precisamente descritas como *arranjos* binários ordenados; cada par de valores na "relação" podendo ser endereçado via (uma forma codificada de) sua posição na sequência. Para isso, cada par tem um identificador implícito. Os domínios são representados por "conjuntos de valores", isto é, arranjos unários ordenados nos quais, novamente, cada elemento possui um identificador implícito, que é a sua posição. Os valores de dados em um dado par dentro de qualquer dos arranjos binários são identificadores, seja de elementos em conjuntos de valores ou de outros pares. Assim, por exemplo, a tripla < 'S1', 'P1', 300 > da relação SP

poderia ser representada como o par de valores $\langle a, b \rangle$, onde b é o identificador do valor 300 no conjunto de valores QTY e a é o identificador do par $\langle c, d \rangle$ no arranjo binário S# – P#, e c e d , por sua vez, são respectivamente identificadores de S1 e P1 nos conjuntos de valores S# e P#. Os dois conjuntos de valores e os arranjos binários empregam técnicas simples porém efetivas de compressão – foi obtida uma redução de espaço de cerca de 70 por cento em uma aplicação estudada (uma estrutura de listas de material). Foram também usadas técnicas de arquivo diferencial [2.24]. Para acesso ao banco de dados, o sistema fornece uma coleção de operadores de conjuntos, principalmente “intercalação”, que pode ser considerada como uma junção seguida de uma projeção. Os tempos de acesso pareceram razoavelmente aceitáveis. O artigo conclui com algumas observações interessantes sobre aspectos de confiabilidade, segurança e integridade desses sistemas.

12.25 E. H. Beitz. “A Set-Theoretic View of Data-Base Representation.” *Proc. 1974 ACM SIGMOD Workshop on Data Description Access and Control*.

12.26 E. H. Beitz. “Sets as a Model for Data Base Representation: Much Ado About Something”. *Proc. ACM Pacific Conference, San Francisco* (abril de 1975). Disponível na ACM Golden Gate Chapter, P.O. Box 24055, Oakland, California 94623.

Esses dois artigos [12.25, 12.26], bem como [12.24], não estão realmente voltados para a visão do usuário do banco de dados, mas sim para os níveis conceitual e interno do sistema. Entretanto, a estrutura conceitual proposta poderia ser perfeitamente adequada para suportar visões relacionais. Cada entidade e cada “propriedade” (tal como peso = 17) recebem um identificador único do sistema; o modelo conceitual então consiste de (a) o conjunto de todos os identificadores de entidades, (b) o conjunto de todos os identificadores de propriedades, e (c) uma relação binária definida sobre (a) e (b). No nível de armazenamento, esta relação binária é representada duas vezes, uma em cada direção, dando simetria de acesso e, incidentalmente, uma cópia automática de descarga. Cada uma das duas representações consiste de um índice no que for apropriado – (a) ou (b). São declaradas muitas vantagens sobre esta abordagem.

13

Cálculo Relacional

13.1 INTRODUÇÃO

A idéia de cálculo em predicado como base para uma linguagem de consulta parece ter se originado em um artigo de Kuhns [13.1]. O conceito de um cálculo *relacional* – isto é, um cálculo predicado aplicado, especificamente talhado para bancos de dados relacionais – foi primeiramente proposto por Codd em [12.1]; uma linguagem explicitamente baseada neste cálculo, a Sublinguagem de Dados ALPHA (DSL ALPHA), foi também apresentada por Codd [13.2]. A DSL ALPHA propriamente nunca foi implementada, mas uma linguagem muito semelhante a ela em seu espírito, chamada QUEL, foi usada como linguagem de consulta no sistema INGRES [13.7–13.16]. Examinaremos o QUEL com mais alguns detalhes posteriormente. Registraremos também que tanto a SQL como o QBE incorporaram alguns elementos do cálculo.

Um aspecto fundamental do cálculo de [12.1], e de linguagens nele baseadas, é a noção de *variável tupla*. Uma variável tupla é uma variável que “se estende” sobre alguma relação com nome¹ – isto é, uma variável cujos únicos valores permitidos são tuplas daquela relação. (Em outras palavras, se a variável tupla T se estende sobre uma relação R, então, em qualquer instante, T representa alguma tupla individual de R.) Por exemplo, a consulta “Obtenha os números dos fornecedores de Londres” pode ser expressa em QUEL como se segue:

```
RANGE OF SX IS S  
RETRIEVE (SX.S#) WHERE SX.CITY = 'LONDON'
```

A variável tupla aqui é SX, que se estende sobre a relação S. A consulta pode ser parafraseada: ‘Para cada valor possível da variável SX, recupere o componente S# daquele valor, se e somente se o componente CITY tiver o valor ‘LONDON’.’

1

Mais precisamente, uma variável tupla se estende ou sobre uma relação com nome ou uma *união* de duas ou mais relações com nome (compatíveis de união). Para simplificar a discussão, ignoraremos o caso de união na maior parte deste capítulo.

[Aproveitando a oportunidade, observemos que a formulação SQL desta consulta – SELECT S# FROM S WHERE CITY = 'LONDON' – não requer a introdução explícita de uma variável tupla, fazendo ao invés disso com que o nome de relação S cumpra o papel implicitamente. Mas o conceito básico é o mesmo. Para se entender como é avaliada a consulta SQL, é necessário imaginar-se a *variável tupla* S se estendendo sobre a *relação* S. Com efeito, a SQL simplesmente possui uma regra implícita para a definição de variáveis tupla que é adequada a casos simples. Em situações mais complexas, o usuário tem que introduzir variáveis tupla explicitamente; veja, por exemplo, os Exemplos 7.2.6, 7.2.13 e 7.2.19.]

Por estar fundamentado em variáveis tupla (e para distingui-lo do cálculo de domínio discutido abaixo), o cálculo relacional original de [12.1] tornou-se conhecido como o cálculo de *tupla*. O cálculo de tupla está descrito na Seção 13.2.

Mais recentemente, Lacroix e Pirotte [13.3] propuseram um cálculo relacional alternativo, o cálculo de *domínio*, no qual as variáveis tupla foram substituídas por variáveis domínio – isto é, variáveis que se estendem sobre os domínios básicos ao invés de sobre relações. Uma linguagem chamada IIL, baseada nesse cálculo, é apresentada pelos mesmos autores em [13.4]. O Query By Example também pode ser considerado como uma implementação do cálculo de domínio. (Na versão QBE do exemplo QUEL acima,

S	S#	SNAME	STATUS	CITY
	P.SX			LONDON

o elemento de exemplo SX é uma variável que se estende sobre o domínio de números de fornecedores.) DEDUCE [7.6] é outro exemplo de linguagem orientada para domínio. Nós iremos discutir o cálculo de domínio na Seção 13.3.

13.2 CÁLCULO RELACIONAL ORIENTADO PARA TUPLA

A construção primária do cálculo relacional orientado para tupla (cálculo de tupla, de forma abreviada) é a *expressão do cálculo de tupla*. Uma expressão de cálculo de tupla é essencialmente uma definição não procedural de alguma relação em termos de algum conjunto dado de relações. Uma expressão como essa pode portanto claramente ser usada para definir o resultado de uma consulta, ou o objetivo de uma atualização, ou uma visão (no sentido do Sistema R), e assim por diante. Nesta seção apresentaremos uma definição bastante formal do cálculo de tupla, seguindo as linhas de [12.1] (embora simplificada), prosseguindo depois com uma série de exemplos.

As expressões do cálculo de tupla são construídas a partir dos seguintes elementos.

- *Variáveis tupla* T, U, V, . . . Cada variável tupla fica limitada a se estender sobre alguma relação com nome². Se a variável tupla T representa a tupla t (a qualquer momento), então a expressão T.A representa o componente A de t (naquele momento), onde A é um atributo da relação sobre a qual T se estende.

²

Ou sobre uma união (veja nota 1 de rodapé na pág. 233). No caso de união, as relações envolvidas têm que ser não só compatíveis de união, mas também ter nomes de atributos idênticos.

- Condições da forma $x * y$, onde * é qualquer dos $=, \neg =, <, \leq, >, \geq$, ou \neq , e pelo menos um dos x e y é uma expressão da forma T.A (veja o parágrafo prévio) e o outro é ou uma expressão similar ou uma constante.
- Fórmulas bem-formadas (WFFs). Uma WFF é uma construção a partir de condições, operadores Booleanos (AND, OR, NOT), e quantificadores (\exists , \forall) de acordo com as regras F1–F5 abaixo.

- F1. Cada condição é uma WFF.
- F2. Se f é uma WFF, então também o são (f) e NOT (f).
- F3. Se f e g são WFFs, então também o são (f AND g) e (f OR g).³
- F4. Se f é uma WFF na qual T ocorre como uma variável livre (veja abaixo), então $\exists T(f)$ e $\forall T(f)$ são WFFs.
- F5. Nada mais é uma WFF.

Variáveis Livres e Limitadas

Cada ocorrência de uma variável tupla dentro de uma WFF é *livre* ou *limitada*. [Uma “ocorrência” de uma variável tupla é o aparecimento do nome da variável dentro da sequência de símbolos que é a WFF em consideração. Uma variável tupla ocorre dentro de uma WFF no contexto de uma expressão T.A (onde T é a variável tupla e A é um atributo da relação associada), ou como a variável que se segue a um dos símbolos quantificadores \exists , \forall].

1. Dentro de uma condição, todas as ocorrências de variáveis tupla são livres.
2. Ocorrências de variáveis tupla nas WFF (f), NOT(f) são livres/limitadas dependendo de serem elas livres/limitadas em f . Ocorrências de variáveis tupla nas WFFs (f AND g), (f OR g) são livres/limitadas dependendo de serem elas livres/limitadas em f ou g (quaisquer f , g em que elas apareçam).
3. Ocorrências de T que sejam livres em f são limitadas nas WFFs $\exists T(f)$, $\forall T(f)$. Outras ocorrências de tuplas em f serão livres/limitadas nessas WFFs dependendo de serem livres/limitadas em f .

Finalmente, uma *expressão de cálculo de tupla* é uma expressão da forma:

T.A, U.B, . . . , V.C [WHERE f]

onde T, U, . . . , V são variáveis tupla; A, B, . . . , C são atributos das relações associadas; e f é uma WFF contendo exatamente T, U, . . . , V como variáveis livres. O valor desta expressão é uma projeção do subconjunto do produto Cartesiano $T \times U \times \dots \times V$ (onde T, U, . . . , V se estendem por todos os seus valores possíveis) no qual f é avaliado como *verdadeiro* – ou, se “WHERE f ” for omitido, uma projeção de todo o produto Cartesiano. A projeção em questão é naturalmente obtida dos componentes (atributos) indicados pelas entradas na lista T.A, B. U, . . . , V.C (a “lista objetivo”).

Exemplos

Nestes exemplos usaremos as variáveis tupla SX, SY, . . . da relação S; PX, PY, . . . da relação P; SPX, SPY, . . . da relação SP.

³ As convenções usuais foram adotadas para se eliminar os parêntesis desnecessários.

Algumas condições válidas

```
SX.S# = 'S1'  
SX.S# = SPY.S#  
SPY.P# ≠ PZ.P#
```

Algumas WFF válidas

```
NOT (SX.CITY = 'LONDON')  
SX.S# = SPY.S# AND SPY.P# ≠ PZ.P#  
ESPX (SPX.S# = SX.S# AND SPX.P# = 'P2')
```

O quantificador existencial \exists é lido como “existe”. Assim, o último exemplo deve ser lido como “Existe uma tupla SP com um valor S# igual ao componente S# de SX [qualquer que ele seja] e um valor P# igual a P2”.

```
∀PZ (PZ.COLOR = 'RED')
```

O símbolo \forall representa o quantificador *universal* (“para todos”). Portanto esta WFF deve ser lida como “Para todas as tuplas P, a cor é vermelha”. O quantificador universal está incluído por pura conveniência; ele não é essencial — a identidade

```
vx(f) = NOT (ex (NOT f))
```

mostra que qualquer WFF envolvendo \forall pode sempre ser substituída por uma WFF equivalente envolvendo \exists . Por exemplo, a afirmativa (verdadeira) “para todo inteiro x existe um inteiro y tal que $y > x$ ” é equivalente à afirmativa “não há inteiro x para o qual não existe um inteiro y tal que $y > x$ ”. Mas é freqüentemente mais fácil pensar-se em termos de \forall ao invés de \exists com uma dupla negativa.

Variáveis livres e limitadas

SX, SPY, e PZ são todas livres nas primeiras duas WFFs acima. Na terceira WFF, SX é livre e SPX é limitada. De forma semelhante, PZ é limitada na quarta WFF. Para examinar o conceito de variáveis limitadas mais de perto, vamos usar um exemplo mais simples:

```
ex (x > 3)
```

(onde x se estende sobre o conjunto de inteiros). A variável limitada x nesta fórmula faz o papel de *fictícia* — ela serve somente para unir a expressão entre parêntesis ao quantificador externo. A fórmula simplesmente estabelece que existe algum inteiro, digamos x , que é maior do que três. O significado da fórmula não se alteraria se todas as ocorrências de x fossem substituídas por alguma outra variável y [dando $\exists y(y > 3)$].

Consideremos agora

```
ex (x > 3) AND x < 0
```

Aqui há três ocorrências de x , *referenciando a duas variáveis diferentes*. As primeiras duas referências são limitadas, e poderiam ser substituídas por alguma outra variável sem altera-

ção do significado. A terceira ocorrência é livre, e *não pode* ser substituída impunemente. Por isso, das duas expressões abaixo, a primeira é equivalente à acima, e a segunda não:

```
 3 y (y > 3) AND x < 0  
 3 y (y > 3) AND y < 0
```

Expressões de cálculo de tupla

```
SX.S#  
SX.S# WHERE SX.CITY = 'LONDON'  
SX.S#, SX.CITY WHERE 3SPX (SPX.S# = SX.S# AND  
SPX.P# = 'P2')
```

A primeira delas denota o conjunto de todos os números de fornecedores na relação S; a segunda denota o subconjunto de todos os números de fornecedores para os quais a cidade é Londres. A terceira é uma representação de cálculo de tupla da consulta: "Obtenha os números e as cidades dos fornecedores que fornecem a peça P2."

O cálculo de tupla é usado como uma base para a definição de ser relationalmente completa dada em [12.1]. O artigo forneceu também um algoritmo para a conversão de uma expressão arbitrária de cálculo em uma expressão equivalente da álgebra relacional (mostrando assim que a álgebra é completa). Por outro lado, Ullman [13.6] mostra que qualquer expressão algébrica pode ser convertida em uma expressão de cálculo de tupla, sendo assim as duas linguagens formalmente equivalentes.

QUEL

Vamos concluir esta seção com alguns exemplos de cálculo de tuplas da linguagem QUEL. Primeiramente daremos as definições de todas as variáveis tupla que precisaremos.

```
A FAIXA DE SX É S  
A FAIXA DE PX É P  
A FAIXA DE SPX É SP  
A FAIXA DE SPY É SP
```

13.2.1 – Obtenha os números dos fornecedores de Paris com status > 20 (Exemplo 8.2.3)

```
RETRIEVE (SX.S#) WHERE SX.CITY = 'PARIS' AND  
SX.STATUS > 20
```

Os atributos do resultado podem receber nomes especificados pelo usuário, se desejado. Por exemplo,

```
RETRIEVE (FOURNISSEUR = SX.S#)  
WHERE SX.CITY = 'PARIS' AND  
SX.STATUS > 20
```

13.2.2 Obtenha os números e cidades dos fornecedores que fornecem a peça P2.

```
RETRIEVE (SX.S#, SX.CITY) WHERE SX.S# = SPX.S# AND  
SPX.P# = 'P2'
```

Vimos anteriormente que SPX tem que ser existencialmente quantificada neste exemplo. Entretanto, em QUEL, todas as variáveis tupla que aparecem depois de WHERE mas não na lista objetivo antes de WHERE estão *supostamente* existencialmente quantificadas. Isto é, a fórmula que se segue ao WHERE é suposta estar na forma normal prenex (veja exemplo 13.2.3 abaixo) com todos os quantificadores omitidos. Os quantificadores nunca são estabelecidos explicitamente em QUEL.

13.2.3 Obtenha os nomes dos fornecedores que fornecem pelo menos uma peça vermelha (Exemplo 7.2.10).

Em cálculo “puro” de tupla,

```
SX.SNAME WHERE  $\exists$ SPX (SX.S# = SPX.S# AND  
 $\exists$ PX (SPX.P# = PX.P# AND  
PX.COLOR = 'RED'))
```

Ou equivalente (mas na *forma normal prenex*, na qual todos os quantificadores aparecem na frente da WFF),

```
SX.SNAME WHERE  $\exists$ SPX ( $\exists$ PX (SX.S# = SPX.S# AND  
SPX.P# = PX.P# AND  
PX.COLOR = 'RED'))
```

Em QUEL,

```
RETRIEVE (SX.SNAME) WHERE SX.S# = SPX.S# AND  
SPX.P# = PX.P# AND  
PX.COLOR = 'RED'
```

13.2.4 Obtenha os nomes dos fornecedores que fornecem pelo menos uma peça suprida pelo fornecedor S2. (Exemplo 7.2.12)

```
RETRIEVE (SX.SNAME) WHERE SX.S# = SPX.S# AND  
SPX.P# = SPY.P# AND  
SPY.S# = 'S2'
```

13.2.5 Para cada peça fornecida, obtenha o número da peça e os nomes de todas as cidades que fornecem a peça. (Exemplo 7.2.5)

```
RETRIEVE (SPX.P#, SX.CITY) WHERE SPX.S# = SX.S#
```

EM SQL, o usuário pode especificar UNIQUE para eliminar duplicatas. Em QUEL, as duplicatas serão eliminadas se o usuário guardar o resultado do RETRIEVE no banco de dados (especificando “INTO nome-da-relação” após a palavra chave RETRIEVE), mas não se o resultado for simplesmente exposto no terminal.

13.2.6 Obtenha os nomes dos fornecedores que não fornecem a peça P2. (Exemplos 7.2.14 e 7.2.17)

Em "puro" cálculo de tupla,

```
SX.SNAME WHERE NOT ( $\exists$ SPX (SPX.S# = SX.S# AND  
SPX.P# = 'P2'))
```

Ou equivalentemente,

```
SX.SNAME WHERE  $\forall$ SPX (SPX.S#  $\neq$  SX.S# OR  
SPX.S#  $\neq$  'P2')
```

Em QUEL,

```
RETRIEVE (SX.SNAME) WHERE  
COUNT(SPX.S# WHERE SPX.S# = SX.S# AND  
SPX.P# = 'P2')  
= 0
```

QUEL não suporta o quantificador universal. Entretanto, fornece o arranjo usual de funções integradas, de tal forma que o efeito do quantificador universal pode ser obtido como indicado no exemplo. (A afirmativa "Para todos os x , f é verdadeiro" é equivalente à afirmativa "A contagem da quantidade de x em que f é falso é zero").

Observe que a instrução QUEL

```
RETRIEVE SX.SNAME WHERE NOT (SPX.S# = SX.S# AND  
SPX.P# = 'P2')
```

não obtém o resultado desejado. (Por que não? O que ela faz?)

13.2.7 Aumente a quantidade de embarques em 100 para todos os fornecedores de Londres.

```
REPLACE SPX (QTY = QTY + 100) WHERE SPA.S# = SX.S# AND  
SX.CITY = 'LONDON'
```

13.2.8 Adicione a peça P7 (nome 'WASHER', cidade 'ATHENS', outros valores desconhecidos) à tabela P. (Veja Exemplo 7.4.5.)

```
APPEND TO P (P# = 'P7', PNAME = 'WASHER', CITY = 'ATHENS')
```

Na nova tupla, o peso (WEIGHT) é colocado como zero, e cor (COLOR) é colocada como uma seqüência vazia (comprimento zero).

13.2.9 Remova todos os embarques do fornecedor S1.

```
DELETE SPX WHERE SPX.S# = 'S1'
```

Maiores detalhes sobre INGRES e a linguagem QUEL encontram-se nas referências [13.7–13.16].

13.3 CÁLCULO RELACIONAL ORIENTADO PARA DOMÍNIO

Como mencionamos na Seção 13.1, o cálculo relacional orientado para domínio (cálculo de domínio, resumidamente) difere do cálculo de tupla por ter suas variáveis se estendendo sobre domínios ao invés de relações. As expressões do cálculo de domínio são construídas a partir dos seguintes elementos:

- *Variáveis domínio* D, E, F, . . . Cada uma restrita a se estender sobre um domínio específico. (“Variável elemento” seria um nome melhor, pois os valores são *elementos* do domínio, não domínios.)
- *Condições*, que podem ter duas formas: (a) comparações simples da forma $x * y$, como no cálculo de tupla, exceto que x e y são agora variáveis domínio (ou constantes); e (b) condições de membros associados, da forma R (termo, termo . . .). Aqui R é uma relação, e cada “termo” é um par A:V, onde A é um atributo de R e V é ou uma variável domínio ou uma constante. Por exemplo, SP (S#: ‘S1’, P#: ‘P1’) é uma condição de membro associado (que é avaliada como *verdadeira* se e somente se existir uma tupla SP possuindo S# = ‘S1’ e P# = ‘P1’).
- *Fórmulas bem-formadas* (WFFs), formadas de acordo com as regras F1–F5 da Seção 13.2 (mas com a definição revisada de “condição”).

Variáveis livres e limitadas

As regras referentes a variáveis livres e limitadas dadas para o cálculo de tupla aplicam-se também ao cálculo de domínio, dentro desse contexto. Observe em particular que a variável x em “ $\exists x$ ” e “ $\forall x$ ” é agora uma variável *domínio*.

Uma expressão do cálculo de domínio é portanto uma expressão da forma

D, E, . . . , F [WHERE f]

onde D, E, . . . , F são variáveis domínio, e f é uma WFF contendo exatamente D, E, . . . , F como variáveis livres. O valor desta expressão é o sobconjunto do produto Cartesiano $D \times E \times \dots \times F$ (onde D, E, . . . , F se estendem sobre todos os seus possíveis valores) para o qual f é avaliado como *verdadeiro* — ou, se “WHERE f” estiver omitido, todo o produto Cartesiano.

Exemplos

Nestes exemplos usaremos variáveis domínio com nomes formados pela adição de X, Y, Z ao nome apropriado de domínio — exceto que, para domínios cujos nomes terminem por “#”, retiraremos o “#”. Lembramos ao leitor que no banco de dados de fornecedores e peças cada atributo tem o nome do seu domínio básico.

Algumas condições válidas

```
CITYX := 'LONDON'  
S (CITY : 'LONDON')  
S (CITY : CITYX)  
SP (S# : SY, P# : PZ)
```

Como mencionado anteriormente, uma condição de membro associado é avaliada como *verdadeira* se e somente se existir uma tupla da relação especificada possuindo os valores dos atributos especificados.

Algumas WFFs válidas

```
NOT S (CITY : 'LONDON')
```

Isto será *verdadeiro* se e somente se não houver uma tupla S contendo 'LONDON' como valor de cidade. Em contraste

```
S (CITY : CITYX) AND CITYX ≠ 'LONDON'
```

Isto será *verdadeiro* se e somente se existir uma tupla S contendo 'LONDON' como valor de cidade.

```
S (S# : SX, CITY : CITYX) AND SP (S# : SX, P# : 'P2')
```

Esta WFF pode ser lida "SX e CITYX são o número do fornecedor e a cidade do fornecedor que fornece a peça P2".

```
∃CITYX (S (CITY : CITYX) AND P (CITY : CITYX))
```

"Existe pelo menos uma cidade possuindo tanto um fornecedor quanto uma peça ali localizados."

```
∀CITYX (S (CITY : CITYX))
```

"Toda cidade conhecida pelo sistema tem pelo menos um fornecedor correspondente."

Expressões do cálculo de domínio

```
.SX  
SX WHERE S (S# : SX)  
SX WHERE S (S# : SX, CITY : 'LONDON')  
SX, CITYX WHERE S (S# : SX, CITY : CITYX) AND  
SP (S# : SX, P# : 'P2')
```

A primeira destas denota o conjunto de todos os números de fornecedores; a segunda denota o conjunto de todos os números de fornecedores na relação S; a terceira denota o conjunto de todos os números de fornecedores da relação S localizados em Londres. A quarta é uma representação de cálculo de domínio da consulta "Obtenha os números e as cidades dos fornecedores que fornecem a peça P2" (observe que a versão de cálculo de tupla desta consulta exigiu um quantificador existencial).

```
SX, CITYX WHERE S (S# : SX, CITY : CITYX) AND  
∃PX (SP (S# : SX, P# : PX) AND  
P (P# : PX, COLOR : 'RED'))
```

"Obtenha os números e as cidades dos fornecedores que fornecem pelo menos uma peça vermelha."

Para exemplos de uma linguagem que é pelo menos parcialmente uma implementação do cálculo de domínio, veja o Capítulo 11 (Query By Example). A linguagem ILL de Lacroix e Pirotte [13.3, 13.4] não será discutida aqui; ILL é mais profundamente ligada ao cálculo de domínio do que QUEL ao cálculo de tupla, e exemplos iriam requerer muito mais explanações preliminares. Recomendamos que o leitor interessado estude as referências.

A diferença entre cálculo de tupla e cálculo de domínio reside basicamente na forma como o usuário percebe o banco de dados. No caso do banco de dados de fornecedores e peças, o cálculo de tupla encoraja o usuário a pensar em termos de três tipos de entidades (fornecedores, peças, embarques), cada uma possuindo diversas propriedades. Em contraste, o cálculo de domínio encoraja o usuário a pensar em termos de uma quantidade maior de tipos de entidades (fornecedores, peças, cidades, cores, quantidades, ...), e ver as três relações S, P, e SP como representando várias associações entre esses tipos de entidades. A formulação de uma dada consulta no cálculo de domínio tende a ser algo mais simples do que a formulação no cálculo de tupla (particularmente se permitirmos a omissão dos quantificadores, como em QUEL), pela seguinte razão: se uma dada entidade, digamos um fornecedor, ocorrer diversas vezes na instrução da consulta em inglês, a formulação no cálculo de domínio conterá diversas ocorrências da mesma variável de domínio correspondente; na formulação no cálculo de tupla, conterá ocorrências de diversas variáveis tupla distintas (tais como SX, SPX), com "condições de junção" (tais como SX.S#=SPX.S#) conectando essas ocorrências.

Finalmente, é fácil ver-se que o cálculo de domínio é relationalmente completo. Nós já mencionamos a prova de Ullman [13.6] de que qualquer expressão da álgebra relacional é equivalente a uma expressão do cálculo de tupla. Ele também mostra [13.6] que qualquer expressão do cálculo de tupla pode ser convertida em uma expressão equivalente do cálculo de domínio, e que qualquer expressão do cálculo de domínio pode ser convertida em uma expressão algébrica equivalente. As três linguagens são portanto equivalentes umas às outras em suas potências de seleção.

EXERCÍCIOS

13.1 Seja $f(x, y)$ uma fórmula arbitrária com variáveis livres x e y . Quais das seguintes instruções são verdadeiras?

- $\exists x \ (\exists y \ (f(x, y))) \equiv \exists y \ (\exists x \ (f(x, y)))$
- $\forall x \ (f(x, y)) \equiv \text{NOT } \exists x \ (\text{NOT } f(x, y))$
- $\exists x \ (\forall y \ (f(x, y))) \equiv \forall y \ (\exists x \ (f(x, y)))$
- $\forall x \ (\forall y \ (f(x, y))) \equiv \forall y \ (\forall x \ (f(x, y)))$

13.2 Dê as soluções de cálculo de tupla e cálculo de domínio para os exercícios 7.1–7.28. Nota: Para simplificar algumas das soluções, é conveniente introduzir o operador de implicação lógica "IF ... THEN ...". "IF A THEN B" é definido como sendo equivalente a "(NOT A) OR B".

13.3 Suponha que se introduza uma regra semelhante (mas não idêntica) à da QUEL, com o seguinte efeito: Na fórmula que se segue ao WHERE, é assumido um quantificador existencial (na extremidade esquerda) para qualquer variável que não apareça na lista objetivo e não quantificada explicitamente. Reveja suas soluções do Exercício 13.2 para tirar partido desta regra.

13.4 Mostre que qualquer expressão do cálculo de tupla tem uma equivalente no cálculo de domínio (e vice-versa).

13.5 Forneça equivalentes em cálculo de tupla e cálculo de domínio para cada um dos cinco operadores primitivos algébricos.

REFERÊNCIAS E BIBLIOGRAFIA

13.1 J. L. Kuhns. "Answering Questions by Computer; A Logical Study," Report RM-5428-PR, Rand Corp., Santa Monica, Calif. (1967).

13.2 E. F. Codd. "A Data Base Sublanguage Founded on the Relational Calculus," *Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control*.

13.3 M. Lacroix and A. Pirotte. "Domain-Oriented Relational Languages." *Proc. 3rd International Conference on Very Large Data Bases* (outubro de 1977).

13.4 M. Lacroix and A. Pirotte. "ILL: An English Structured Query Language for Relational Data Bases." In *Architecture and Models in Data Base Management Systems* (ed., G. M. Nijssen). North-Holland (1977).

13.5 A. Pirotte and P. Wodon. "A comprehensive Formal Query Language for a Relational Data Base." *R.A.I.R.O. Informatique/Computer science* 11, nº 2 (1977).

A FQL, como a ILL, é baseada no cálculo de domínio, mas é muito mais formal (menos "parecida com o inglês") do que a ILL.

13.6 J. D. Ullman. *Principles of Database Systems*. Washington, D.C.: Computer Science Press (1979).

13.7 G. D. Held, M. R. Stonebraker, and E. Wong. "INGRES - A Relational Data Base System." *Proc. NCC* 44 (maio de 1975).

Fornece uma descrição em alto nível do sistema; inclui uma definição de QUEL.

13.8 M. R. Stonebraker, E. Wong, and P. Kreps. "The Design and Implementation of INGRES." *ACM Transactions on Database Systems* 1, nº 3 (setembro de 1976).

Uma versão expandida de [13.7].

13.9 M. R. Stonebraker. "Getting Started in INGRES - A Tutorial." Berkeley: University of California, Electronics Research Laboratory Memorandum ERL-M518 (abril de 1975).

13.10 R. Epstein. "Creating and Maintaining a Database Using INGRES." Berkeley: University of California, Electronics Research Laboratory Memorandum UCB/ERL M77/71 (Dezembro de 1977).

13.11 E. Allman, G. D. Held, and M. R. Stonebraker. "Embedding a Data Manipulation Language in a General Purpose Programming Language." *Proc. 1976 ACM SIGPLAN/SIGMOD Conference on Data Abstraction* (março de 1976).

Descreve um pré-compilador operacional para EQUEL (QUEL embutida na linguagem de programação C).

13.12 G. D. Held and M. R. Stonebraker. "Storage Structures and Access Methods in the Relational Data Base Management System INGRES." *Proc. ACM Pacific Conference, San Francisco*, (abril de 1975). Disponível da ACM Golden Gate Chapter, P.O. Box 24055, Oakland, California 94623.

Como o Sistema R, o INGRES inclui um "interface do registro armazenado" chamado AMI (análogo ao RSI). Neste nível, as relações são representadas como arquivos armazenados, com cada tupla representada como um registro armazenado. As tuplas têm TIDs, como no RSS. Um determinado arquivo armazenado pode ter qualquer uma das seguintes estruturas de armazenamento: (1) *acumulado* (uma tabela não ordenada); (2) *rândômica*; (3) *isam* (índice na chave primária). Diversas formas de (2) e (3), incluindo algumas técnicas de compressão envolvidas, estão discutidas neste artigo. A estrutura de armazenamento específica usada para uma determinada relação não aparece para os componentes de mais alto nível do sistema; o AMI fornece operadores genéricos INSERT, REPLACE/DELETE/GET (dada uma TID), FIND (para estabelecer as condições de início e fim de uma pesquisa sequencial), e GET (dentro da pesquisa), que operam sobre qualquer estrutura. Em adição, o componente de nível mais alto do sistema pode criar índices secundários sobre qualquer relação; um índice secundário é em si tratado como uma relação (armazenado como um arquivo *isam*), na qual um campo fornece indicadores de localização (TIDs) dos registros da relação indexada (mas, naturalmente, esses TIDs não

estão acessíveis ao usuário INGRES). A AMI propriamente desconhece um relacionamento entre uma relação de índice secundário e a correspondente relação indexada.

13.13 E. Wong and K. Youssefi. "Decomposition – A Strategy for Query Processing." *ACM Transactions on Data Base Systems* 1, nº 3 (setembro de 1976).

Descreve a estratégia para o processamento de consultas em INGRES. O procedimento geral é o de quebrar a consulta envolvendo mais de uma variável tupla em uma sequência de consultas envolvendo uma daquelas variáveis cada, usando alternativamente *redução* e *substituição de tupla* para conseguir a decomposição desejada. *Redução* é o processo de remover da consulta um componente que tenha exatamente uma variável em comum com o restante da consulta. A *substituição de tupla* é o processo de substituir uma das variáveis tupla de cada vez. Este artigo fornece algoritmos para a redução e seleção de variáveis para a substituição de tupla.

13.14 P. Hawthorne and M. R. Stonebraker. "The Use of Technological Advances to Enhance Data Management System Performance." *Proc. 1979 ACM SIGMOD International Conference on Management of Data* (maio de 1979).

13.15 R. Epstein. "Techniques for Processing of Aggregates in Relational Database Systems." Berkeley: University of California, Electronics Research Laboratory Memorandum UCB/ERL M79/8 (fevereiro de 1979).

Os "agregados" do título se referem a funções integradas tais como COUNT.

13.16 M. R. Stonebraker. "Requiem for a Data Base System." Berkeley: University of California, Electronics Research Laboratory Memorandum UCB/ERL M79/4 (janeiro de 1979).

Um depoimento sobre a história do projeto INGRES (até janeiro de 1979). A ênfase é nos erros e lições aprendidas, mais do que nos sucessos.

13.17 J. B. Rothnie, Jr. "Evaluating Inter-Entry Retrieval Expressions in a Relational Data Base Management System." *Proc NCC* 44 (1975).

13.18 J. B. Rothnie, Jr. "An Approach to Implementing a Relational Data Management System." *Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control*.

Estes dois artigos [13.17, 13.18] descrevem algumas técnicas usadas no sistema experimental DAMAS (construído no M.I.T.) para a implementação de uma linguagem de recuperação baseada em cálculo. [13.17] é mais um estudo, por sua natureza; [13.18] fornece alguns resultados experimentais e mais detalhes internos. O artigo discute, especificamente, a implementação de expressões de recuperação envolvendo uma variável tupla singela quantificada existencialmente em termos de expressões mais simples conhecidas como "condições Booleanas primitivas", ou PBCs. Uma PBC é um predicado que pode ser estabelecido como verdadeiro ou falso para uma dada tupla pelo exame isolado daquela tupla – isto é, é um predicado que não envolve quantificadores. Os "módulos de armazenamento" de DAMAS, que são responsáveis pela administração do banco de dados armazenado, suportam diretamente as seguintes operações:

- obtenha a próxima tupla em que P é verdadeiro,
- teste a existência de uma tupla tal que P seja verdadeiro,
- desconsidere todas as tuplas em que P seja verdadeiro,

onde P é uma PBC. Usando essas operações, DAMAS manuseia uma recuperação envolvendo R1 (não quantificado) e R2 (existencialmente quantificado) como se segue. Observe que o objetivo da consulta tem que ser alguma projeção de R1.

Etapa 1. No predicado original, torne todos os termos envolvendo R2 verdadeiros e simplifique. O resultado é uma PBC, digamos PBC1. As tuplas de R1 que não satisfizerem a PBC1 podem ser eliminadas de outras considerações.

Etapa 2. Pegue uma tupla (não eliminada) de R1. Substitua os valores desta tupla no predicado original e simplifique, produzindo PBC2. Existe alguma tupla em R2 tal que PBC2 seja verdadeira?

Etapa 3. (Sim) Obtenha a tupla R2 identificada. Extraia valores objetivo da tupla R1 e some à relação resultado. Construa PBC3, selecionando todas as tuplas R1 que contêm os mesmos va-

lores dos atributos objetivo, e use-a para eliminar de consideração todas as tuplas R1 que geriam duplicatas. (Esta eliminação pode ser executada sempre que uma tupla for adicionada ao resultado.) Substitua também valores da tupla R2 obtida no predicado original e simplifique, produzindo PBC4. Obtenha todas as tuplas R1 que satisfazem a PBC4 e some valores objetivo ao resultado.

Etapa 4. (Não) Construa PBC5, selecionando todas as tuplas R1 que produziriam (Etapa 2) uma PBC para a qual não pode existir uma tupla R2 que a torne verdadeira (porque nenhuma tupla R2 tornou PBC2 verdadeira). Elimine essas tuplas R1.

Etapa 5. Repita a partir da etapa 2 até que não restem tuplas R1.

O projeto do algoritmo apresentado está baseado no princípio de que o máximo possível de informações deve ser obtido de cada acesso ao banco de dados. Na prática, entretanto, pode se tornar *mais* caro eliminar tuplas de consideração (por exemplo) do que simplesmente exáminá-las e rejeitá-las. Por esta razão, certas etapas do algoritmo poderão ser ou não aplicadas em determinada situação. No DAMAS, a escolha de aplicá-las ou não é deixada ao usuário, mas foram dadas algumas sugestões para automatizar a escolha.

14

Normalização Adicional

14.1 INTRODUÇÃO

Até agora nós examinamos diversos aspectos de um sistema de banco de dados em geral, e sistemas relacionais em particular. Verificamos as estruturas globais desses sistemas, linguagens para definição e manipulação de dados na forma relacional, métodos para a implementação dessas linguagens, técnicas para representação das relações no armazenamento e outras. Mas ainda não consideramos uma questão muito fundamental, que é: Dado um corpo de dados para ser representado no banco de dados, como decidimos sobre a forma lógica adequada de estrutura para aqueles dados? Em outras palavras, como decidimos que relações são necessárias e quais deveriam ser os seus atributos? Isto é o problema do *projeto do banco de dados*.

Consideremos novamente o banco de dados de fornecedores e peças. O esquema relacional da Fig. 4.7 traz um sentimento de que está correto; é óbvio que são necessárias três relações (S , P , SP), que (por exemplo) $COLOR$ pertence à relação P , $STATUS$ à relação S , QTY à relação SP , e assim por diante. Mas o que nos leva a saber que as coisas são assim? Podemos entender esta questão observando o que ocorre se mudarmos o projeto. Suponhamos que $STATUS$ seja movido da relação S para a relação SP (intuitivamente o lugar errado para isto, pois $STATUS$ se refere a fornecedores, não a embarques). A Fig. 14.1 mostra uma tabulação parcial desta relação SP revisada (que chamaremos de SP' para evitar confusão).

Fica claro da Fig. 14.1 que SP' envolve um monte de redundâncias – o fato de um determinado fornecedor ter um certo status é estabelecido tantas vezes quantas houver embarques daquele fornecedor. Esta redundância pode levar a problemas. Por exemplo, após uma atualização, o fornecedor $S1$ pode estar mostrado como tendo um status de 20 em uma tupla e um status de 30 em outra. Talvez por isso um bom princípio de projeto seja “um fato em um lugar” (isto é, evitar redundância quando possível).

O tópico deste capítulo, teoria da normalização, é basicamente uma formalização de idéias simples como essa – uma formalização que tem aplicação prática na área de projetos de bancos de dados.

SP'	S#	P#	QTY	STATUS
	S1	P1	300	20
	S1	P2	200	20
	S1	P3	400	20
	S1	P4	100	20

Fig. 14.1 Tabulação parcial da relação SP'

Antes de prosseguir, devemos enfatizar o fato de que o projeto de um banco de dados pode ser uma tarefa extremamente complexa. A teoria da normalização é uma ferramenta útil no processo do projeto, mas *não* é uma panacéia. Qualquer um que projete um banco de dados relacional deve estar familiarizado com as técnicas básicas de normalização descritas neste capítulo, mas certamente nós não estamos sugerindo que o projeto seja baseado somente nos princípios de normalização.

Relembremos do Capítulo 4 que relações em um banco de dados relacional são sempre normalizadas, no sentido de serem elas definidas sobre domínios simples (domínios que só contêm valores atômicos). Na Seção 4.2 nós mostramos através de um exemplo como uma relação não normalizada pode ser reduzida a uma forma normalizada equivalente. A teoria da normalização leva este conceito básico muito mais longe. O ponto fundamental é que uma determinada relação, mesmo estando normalizada, pode ainda possuir algumas propriedades indesejáveis (a relação SP' da Fig. 14.1 é um desses casos); a teoria da normalização permite-nos reconhecer esses casos e mostra como essas relações podem ser convertidas em uma forma mais desejável.

Formas Normais

A teoria da normalização está montada em torno do conceito de *formas normais*. Uma relação é dita estar em uma determinada forma normal se ela satisfizer a um conjunto específico de restrições. Por exemplo, uma relação é dita estar na *primeira forma normal* (abreviadamente 1NF) se e somente se ela satisfizer à restrição de conter somente valores atômicos (portanto toda relação normalizada está na 1NF, o que explica a “primeira”).

Foram definidas numerosas formas normais (veja Fig. 14.2). Codd originalmente definiu a primeira, segunda e terceira formas normais (1NF, 2NF, 3NF) na referência [14.1]. Resumidamente, como a Fig. 14.2 induz, todas as relações normalizadas estão na 1NF; algumas relações 1NF estão também em 2NF; e algumas relações 2NF estão também em 3NF. A motivação por trás das definições é de que 2NF é “mais desejável” do que 1NF, em um sentido a ser explicado, e por sua vez 3NF é mais desejável do que 2NF. Assim, o projetista deverá geralmente escolher relações 3NF ao projetar o banco de dados, ao invés de relações 2NF ou 1NF.¹

¹ Esta afirmativa *não* deve ser tomada como uma lei. Algumas vezes existem boas razões para se desprezar os princípios da normalização (veja a Seção 14.8). A única exigência forte é a de que as relações estejam pelo menos na primeira forma normal.

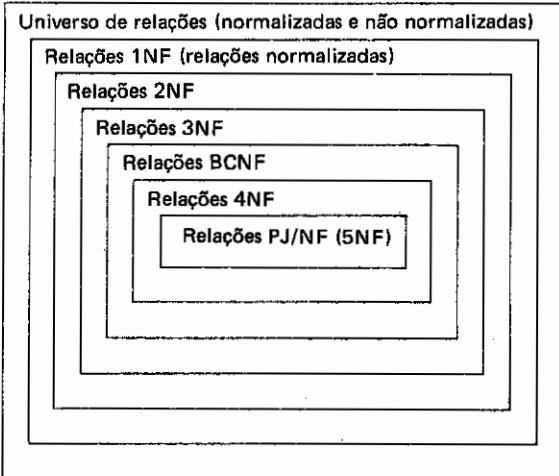


Fig. 14.2 Formas normais.

A definição original de Codd da 3NF [14.1] padecia de certas inadequações, como veremos na Seção 14.4. Uma definição revista (melhor), devida a Boyce e Codd, foi dada em [9.2] — mais forte no sentido de que qualquer relação que fosse 3NF pela nova definição seria certamente 3NF pela antiga, mas uma relação poderia ser 3NF pela definição antiga e não pela nova. A nova 3NF é chamada de Forma Normal de Boyce/Codd (BCNF) para distingui-la da forma anterior. Subseqüentemente, Fagin [14.4] definiu uma nova “quarta” forma normal (4NF) e, mais recentemente, outra forma normal que ele chamou “forma normal projeção-junção” (PJ/NF, também conhecida como 5NF) [14.5]. Como a Fig. 14.2 mostra, algumas relações BCNF estão também na 4NF, e algumas relações 4NF estão também na 5NF.

O leitor deve estar imaginando se não haverá uma 6NF, 7NF, . . . , até o infinito. Embora esta seja uma boa pergunta, nós obviamente ainda não estamos em condições de fazer qualquer consideração detalhada. Vamos nos contentar com a afirmativa bastante equívoca de que existem sem dúvida formas normais adicionais não mostradas na Fig. 14.2, mas que a 5NF é a forma normal “final” em um sentido especial (porém importante). Voltaremos a este tópico no final do capítulo.

Chamamos a atenção do leitor para o fato de que não procuramos ser rigorosos no que se segue; ao invés, confiamos largamente na pura intuição. Sem dúvida, parte da argumentação é que conceitos tais como 4NF, apesar da terminologia algo exótica, são essencialmente muito simples e de bom senso. A maior parte das referências trata do assunto de uma maneira mais formal e rigorosa.

14.2 DEPENDÊNCIA FUNCIONAL

Começaremos introduzindo a noção fundamental de *dependência funcional* (FD).

- Dada uma relação R, o atributo Y de R é *funcionalmente dependente* do atributo

X de R se e somente se cada valor de X em R tiver a ele associado precisamente um valor de Y em R (a qualquer momento).

No banco de dados fornecedores e peças, por exemplo, os atributos SNAME, STATUS, e CITY da relação S são cada um funcionalmente dependentes do atributo S#, pois para cada valor de S# existe precisamente um valor correspondente de SNAME, STATUS e CITY (naturalmente, desde que aquele valor de S# ocorra na relação S). Em símbolos, temos

$$\begin{aligned}S.S\# &\rightarrow S.SNAME \\S.S\# &\rightarrow S.STATUS \\S.S\# &\rightarrow S.CITY\end{aligned}$$

ou mais sucintamente

$$S.S\# \rightarrow S.(SNAME, STATUS, CITY)$$

A afirmativa “ $S.S\# \rightarrow S.CITY$ ” (por exemplo) é lida como “o atributo S.CITY é funcionalmente dependente do atributo S.S#”, ou, equivalentemente, “o atributo S.S# determina funcionalmente o atributo S.CITY”. A afirmativa “ $S.S\# \rightarrow S.(SNAME, STATUS, CITY)$ ” pode ser interpretada de forma semelhante se concordarmos em considerar a combinação (SNAME, STATUS, CITY) como um atributo composto da relação S.

Observe que na definição de dependência funcional não há exigência de que um dado valor de X apareça somente em uma tupla de R. Daremos uma definição alternativa que torna este ponto mais explícito.

- Dada uma relação R, o atributo Y de R é *funcionalmente dependente* do atributo X de R se e somente se, sempre que duas tuplas de R combinarem em seus valores de X, elas também combinem no valor de Y.

Por exemplo, a relação SP' da Fig. 14.1 satisfaz a FD

$$SP'.S\# \rightarrow SP'.STATUS$$

Já vimos que o atributo Y na definição pode ser composto. O mesmo se aplica ao atributo X. Por exemplo, o atributo QTY da relação SP é funcionalmente dependente do atributo composto (S#, P#):

$$SP.(S\#, P\#) \rightarrow SP.QTY$$

(Dada uma combinação particular dos valores S# e P#, existe precisamente um valor QTY — supondo, naturalmente, que a combinação particular (S#, P#) ocorra dentro de SP.)

Uma dependência funcional é uma forma especial de *restrição de integridade*. Quando dizemos, por exemplo, que a relação S satisfaz a FD $S.S\# \rightarrow S.CITY$, estamos dizendo que *qualquer extensão legal* (tabulação) daquela relação satisfaz aquela restrição; em outras palavras, estamos dizendo algo sobre a *intensão* da relação (veja o Capítulo 4). O projeto do banco de dados, por definição, está voltado para intensões ao invés de extensões. Neste capítulo, portanto, iremos usar “relação” para significar a parte intensional da relação, e não sua parte extensional.

É conveniente representar-se as FDs em um determinado conjunto de relações por

meio de um diagrama de dependência funcional.² Um exemplo encontra-se na Fig. 14.3. Observe que na relação S nós temos tanto $S.\# \rightarrow S.SNAME$ como $S.SNAME \rightarrow S.\#$ (porque SNAME é uma chave alternativa da relação S).

Vamos introduzir também o conceito de dependência funcional *total*. O atributo Y é totalmente funcionalmente dependente do atributo X se ele for funcionalmente dependente de X e *não* funcionalmente dependente de qualquer subconjunto apropriado de X (em outras palavras, não existe um subconjunto apropriado X' dos atributos constituintes de X tal que Y seja funcionalmente dependente de X'). Por exemplo, na relação S, o atributo CITY é funcionalmente dependente do atributo composto (S#, STATUS); no entanto, ele não é *totalmente* funcionalmente dependente deste atributo composto porque, naturalmente, ele é funcionalmente dependente também de S# sozinho. (Se Y for funcionalmente dependente de X mas não totalmente, então X tem que ser composto.) No decorrer deste livro usaremos “dependência funcional” com o significado de dependência funcional total, a menos que seja explicitamente estabelecido de outra forma.

Concluiremos esta seção observando que o reconhecimento das dependências funcionais é uma parte essencial do entendimento do significado ou semântica dos dados. O fato de CITY ser funcionalmente dependente de S#, por exemplo, significa que cada fornecedor está localizado precisamente em uma cidade. Em outras palavras, temos uma restrição do mundo real representada no banco de dados, ou seja, que cada fornecedor está localizado precisamente em uma cidade. Como isto é parte da semântica da situação, esta restrição tem que ser observada de alguma forma no banco de dados; a maneira de se garantir que isto seja observado é especificando a restrição na definição do banco de dados (isto é, no esquema conceitual), de forma a que o DBMS possa exigí-la; e a forma de especificá-la no esquema conceitual é declarando a dependência funcional. Mais tarde veremos que os conceitos de normalização levam a meios simples de se declarar essas dependências funcionais.

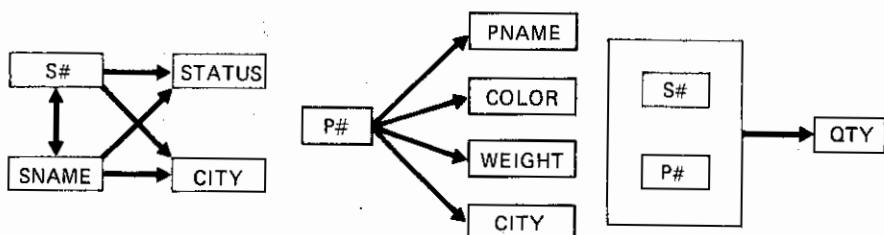


Fig. 14.3 Dependências funcionais nas relações S, P, SP.

2

Todas as dependências consideradas neste capítulo, funcionais ou de outro tipo, estão definidas no contexto de uma relação singela. Também existem dependências que ultrapassam as relações; por exemplo, a restrição de que qualquer valor de S# que apareça na relação SP também apareça na relação S é uma dependência desse tipo. A teoria corrente de normalização fala pouco sobre essas dependências inter-relacionais.

14.3 PRIMEIRA, SEGUNDA E TERCEIRA FORMAS NORMAIS

Estamos agora prontos para descrever 1NF, 2NF, e 3NF. Apresentaremos primeiro uma definição preliminar, *muito* intuitiva, da 3NF para dar uma idéia do ponto que estamos objetivando. Depois consideraremos o processo de se reduzir uma relação arbitrária a uma coleção equivalente de relações 3NF, dando definições algo mais precisas das três formas à medida que avançamos. Entretanto, alertamos de princípio que 1NF, 2NF, e 3NF não são muito significativas por si mesmas, exceto como etapas de suporte para BCNF (e além).

- Uma relação R está na terceira forma normal (3NF) se e somente se, por todo o tempo, cada tupla de R consistir de um valor de chave primária que identifique alguma entidade, juntamente com um conjunto de valores de atributos mutuamente independentes que descrevam aquela entidade de alguma forma.³

Por exemplo, a relação P está na 3NF: cada tupla de P consiste de um valor P#, identificando alguma peça específica, juntamente com quatro partes de informações descritivas referentes à peça – nome, cor, peso e localização. Além disso, cada um dos quatro itens descritivos é independente dos outros três (dois atributos são mutuamente independentes se nenhum for funcionalmente dependente do outro; como sempre, permitimos que os atributos sejam compostos). As relações S e SP também estão na 3NF; as entidades nestes casos são fornecedores e embarques, respectivamente. Em geral, as entidades identificadas pelos valores da chave primária são as entidades fundamentais sobre as quais os dados são gravados no banco de dados (Seção 1.2).

Vamos agora nos voltar para o processo de redução. Primeiro daremos uma definição da primeira forma normal.

- Uma relação está na *primeira forma normal* (1NF) se e somente se todos os domínios básicos contiverem somente valores atômicos.

Esta definição simplesmente afirma que *qualquer* relação normalizada está na 1NF, o que é naturalmente correto. Uma relação que esteja somente na primeira forma normal (isto é, uma relação 1NF que não é também 2NF, é consequentemente não é também 3NF) tem uma estrutura que é indesejável por uma série de razões. Para ilustrar este ponto, suponhamos que as informações referentes a fornecedores e embarques, ao invés de estarem separadas em duas relações (S e SP), sejam reunidas em uma única relação FIRST (S#, STATUS, CITY, P#, QTY). Os atributos aqui têm seus significados usuais; entretanto, para fins de exemplo, introduziremos uma restrição adicional, ou seja – STATUS é funcionalmente dependente de CITY. (O significado desta restrição é que o status de um fornecedor fica determinado pela localização correspondente; por exemplo, todos os fornecedores de Londres têm que ter um status de 20.) Por simplicidade, ignoraremos o atributo SNAME. A chave primária de FIRST é a combinação (S#, P#). A figura 14.4 é o diagrama de dependência funcional para esta relação; observe que o diagrama é “mais complexo” do que um diagrama 3NF.

³

Por simplicidade, estamos supondo no decorrer desta seção que cada relação possui apenas uma chave candidata (isto é, uma chave primária sem chaves alternativas). Em particular, estamos ignorando o atributo SNAME. Esta suposição está refletida nas nossas definições, que (repetimos) não são totalmente rigorosas. O caso de relações possuindo duas ou mais chaves candidatas está discutido na Seção 14.4.

Vemos da Fig. 14.4 que (a) STATUS e CITY não são totalmente funcionalmente dependentes da chave primária, e (b) STATUS e CITY não são mutuamente independentes. São essas duas condições que tornam este diagrama mais complexo do que um diagrama 3NF; e cada uma das duas conduz a problemas. Para ilustrar algumas das dificuldades, consideremos uma tabulação exemplo (extensão) de FIRST (Fig. 14.5). Os valores dos dados mostrados são basicamente os da Fig. 4.7, exceto o status do fornecedor S3, que foi mudado de 30 para 10 para ficar consistente com a nova restrição de que STATUS seja dependente de CITY.

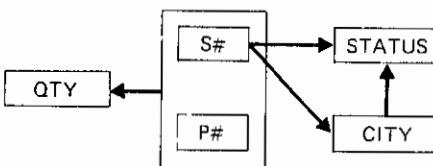


Fig. 14.4 Dependências funcionais na relação FIRST.

A relação FIRST sofre de anomalias com respeito às operações de atualização que são muito semelhantes às encontradas em certas hierarquias (como descrito na Seção 3.3). Para fixar nossas idéias vamos nos concentrar na associação entre fornecedores e cidades – isto é, na dependência funcional de CITY em S#. Ocorrem problemas com cada uma das três operações básicas.

Inserção – Não podemos dar entrada no fato de um determinado fornecedor estar localizado em determinada cidade até que aquele fornecedor forneça pelo menos uma peça. Como podemos ver, a tabulação da Fig. 14.5 não mostra que o fornecedor S5 está localizado em Atenas. A razão é que, até que S5 forneça alguma peça, não teremos o valor apropriado da chave primária. (Lembre-se de que, pela Regra de Integridade 1 (Seção 4.3), nenhum componente do valor da chave primária pode ser nulo; na relação FIRST, os valores de chave primária consistem de um número de fornecedor e um número de peça.)

Remoção – Se removermos de FIRST uma tupla única de um determinado fornecedor, destruiremos não só o embarque conectando aquele fornecedor a alguma peça, mas também a informação de que o fornecedor está localizado em uma determinada cidade. Por exemplo, se removermos de FIRST a tupla S# com valor S3 e P# com valor P2, perdemos a informação de que S3 está localizado em Paris. (Como na Seção 3.3, os problemas de inserção e remoção são na realidade duas faces da mesma moeda.)

Atualização – Em geral, o valor de cidade de um determinado fornecedor aparece várias vezes em FIRST. Esta redundância causa problemas de atualização. Por exemplo, se o fornecedor S1 for deslocado de Londres para Amsterdam, estaremos face ao problema de *ou* pesquisar a relação FIRST para encontrar cada tupla conectando S1 e Londres (e mudá-la), *ou* a possibilidade de produzir um resultado inconsistente (pode constar a cidade de Amsterdam para S1 em um local e Londres em outro).

FIRST	S#	STATUS	CITY	P#	QTY
	S1	20	London	P1	300
	S1	20	London	P2	200
	S1	20	London	P3	400
	S1	20	London	P4	200
	S1	20	London	P5	100
	S1	20	London	P6	100
	S2	10	Paris	P1	300
	S2	10	Paris	P2	400
	S3	10	Paris	P2	200
	S4	20	London	P2	200
	S4	20	London	P4	300
	S4	20	London	P5	400

Fig. 14.5 Tabulação exemplo de FIRST.

A solução para esses problemas é substituir a relação FIRST pelas duas relações SECOND (S#, STATUS, CITY) e SP (S#, P#, QTY). A Fig. 14.6 mostra os diagramas de dependência funcional dessas duas relações; a Fig. 14.7 mostra tabulações de exemplo correspondentes aos valores de dados da Fig. 14.5, exceto que a informação do fornecedor S5 foi agora incorporada à relação SECOND (mas não SP). A relação SP é de fato agora exatamente como a da Fig. 4.7.

Deve ficar claro que esta estrutura revisada evita todos os problemas que tivemos com as operações de atualização envolvendo a associação S#—CITY.

Inserção — Podemos dar entrada na informação de que S5 está localizado em Atenas, mesmo que S5 não esteja correntemente fornecendo peças, simplesmente inserindo a tupla apropriada em SECOND (como na Fig. 14.7).

Remoção — Podemos remover o embarque conectando S3 e P2, removendo de SP a tupla apropriada; não perderemos a informação de que S3 está localizado em Paris.

Atualização — Na estrutura revisada, a cidade de um determinado fornecedor aparece uma vez, não várias (foi eliminada a redundância). Assim, podemos mudar a cidade de S1 de Londres para Amsterdam alterando-a apenas uma vez na tupla relevante de SECOND.

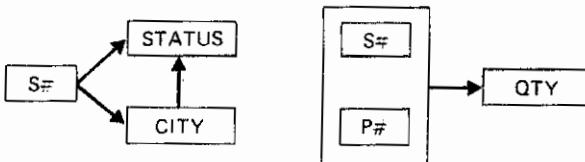


Fig. 14.6 Dependências funcionais nas relações SECOND e SP.

SECOND	S#	STATUS	CITY
	S1	20	London
	S2	10	Paris
	S3	10	Paris
	S4	20	London
	S5	30	Athens

SP	S#	P#	QTY
	S1	P1	300
	S1	P2	200
	S1	P3	400
	S1	P4	200
	S1	P5	100
	S1	P6	100
	S2	P1	300
	S2	P2	400
	S3	P2	200
	S4	P2	200
	S4	P4	300
	S4	P5	400

Fig. 14.7 Tabulações de exemplo de SECOND e SP.

Comparando as Figs. 14.6 e 14.4, nós vemos que o efeito da nossa revisão estrutural foi o de eliminar as dependências funcionais *não totais*, e foi esta eliminação que resolveu as dificuldades. Intuitivamente podemos dizer que na relação FIRST os atributos STATUS e CITY não descrevem a entidade identificada pela chave primária, ou seja, um embarque fornecedor-peça; ao invés disso, eles descrevem somente o fornecedor. A causa dos problemas foi a mistura dos dois tipos de informação na mesma relação.

Daremos agora uma definição da segunda forma normal.⁴

- Uma relação R está na *segunda forma normal* (2NF) se e somente se ela estiver em 1NF e todos os atributos não-chave forem totalmente dependentes da chave primária.

(Um atributo é *não-chave* se ele não participar da chave primária.)⁵ As relações SECOND e SP são ambas 2NF [as chaves primárias são respectivamente S# e a combinação (S#, P#)]. A relação FIRST não é 2NF. Uma relação que esteja na primeira forma normal mas não na segunda pode sempre ser reduzida a uma coleção equivalente de relações 2NF. (Observe, incidentalmente, que uma relação 1NF que não seja também 2NF tem que ter uma chave primária composta.) A redução consiste em substituir a relação por *projeções* adequadas; a coleção dessas projeções é equivalente à relação original, no sentido de que esta relação original sempre poderá ser refeita através da *junção natural* dessas projeções, não havendo assim perda de informação no processo (que é, naturalmente, muito importante). Em outras palavras, o processo é reversível. No nosso exemplo, SECOND e SP são projeções de FIRST, e FIRST é a junção natural de SECOND e SP em S#.

⁴ Veja a nota 3 de rodapé na página 242.

⁵ Veja a nota 3 de rodapé na página 242.

A redução de FIRST no par (SECOND, SP) é um exemplo de *decomposição sem perdas*.⁶ Em geral, dada uma relação R com possíveis atributos compostos A, B, C satisfazendo à FD $R.A \rightarrow R.B$, R poderá sempre ser “decomposta sem perdas” em suas projeções $R1(A, B)$ e $R2(A, C)$ (este teorema foi primeiramente provado por Heath [14.3]). Como não há perda de informação no processo de redução, qualquer informação que possa ser derivada da estrutura original poderá também ser derivada da nova estrutura. No entanto, o inverso não é verdadeiro; a nova estrutura pode conter informações (como o fato de S5 estar localizado em Atenas) que não podiam ser representados na original. Neste sentido, a nova estrutura é um reflexo um pouco mais preciso do mundo real.

Entretanto, a estrutura SECOND/SP ainda causa problemas. A relação SP é satisfatória; na realidade, a relação SP está agora na terceira forma normal, e nós iremos ignorá-la no restante desta seção. A relação SECOND, por outro lado, ainda padece da falta de independência mútua entre seus atributos não-chave. O diagrama de dependência de SECOND ainda é “mais complexo” do que um diagrama 3NF. De forma específica, a dependência de STATUS em S#, embora sendo funcional, é *transitiva* (via CITY): cada valor de S# determina um valor de CITY, e este por sua vez determina o valor de STATUS. Mais uma vez, essa situação transitiva leva a dificuldades nas operações de atualização. (Vamos nos concentrar agora na associação entre valores de cidade e status – isto é, na dependência funcional de STATUS em CITY.)

Inserção — Não podemos dar entrada no fato de uma determinada cidade ter um determinado valor de status – por exemplo, não podemos estabelecer que qualquer fornecedor de Roma tenha um status de 50 – até que tenhamos algum fornecedor localizado naquela cidade. Novamente, a razão é que, até que exista um fornecedor, não teremos o valor apropriado da chave primária.

Remoção — Se removermos uma tupla de SECOND que seja única para determinada cidade, destruiremos não só a informação referente ao fornecedor envolvido, mas também a informação de que aquela cidade tem um determinado valor de status. Por exemplo, se removermos a tupla S5 de SECOND, perderemos a informação de que o valor de status para Atenas é 30. (Outra vez os problemas de inserção e remoção são as duas faces da mesma moeda.)

Atualização — em geral, o valor de status de uma determinada cidade aparece muitas vezes em SECOND (a relação contém ainda alguma redundância). Por isso, se precisarmos modificar o valor de status de Londres de 20 para 30, teremos *ou* o problema de pesquisar a relação SECOND para encontrar todas as tuplas de Londres, *ou* a possibilidade de produzir algum resultado inconsistente (o status de Londres pode constar como sendo 20 em um local e 30 em outro).

6

“Decomposição sem perdas” é um termo bastante estranho. A decomposição em si não pode perder informação; a projeção de uma relação R sobre algum possível atributo composto A contém exatamente a mesma informação do atributo original R.A. Entretanto, a rejunção de duas projeções pode fazer com que a relação R reapareça *juntamente com alguma tupla “espúria” adicional*. Nunca poderá produzir algo que seja *menos* do que R. (*Exercício*: prove esta afirmativa.) Uma decomposição sem perdas garante que a junção produzirá exatamente a relação R. Uma decomposição que não seja sem perdas perde informação no sentido de que a junção poderá produzir uma ampliação da R original, não havendo meios de se saber que tuplas nessa ampliação são espúrias e quais as genuínas.

Novamente a solução para o problema é substituir-se a relação original (SECOND) por duas projeções, neste caso SC(S#, CITY) e CS(CITY, STATUS). A Fig. 14.8 mostra o diagrama de dependência funcional correspondente, e a Fig. 14.9 a tabulação correspondente dos valores de dados da SECOND original (Fig. 14.7).

Outra vez o processo é reversível, pois SECOND é a junção de SC e CS em CITY.

Deve ficar claro que esta nova estrutura evita todos os problemas de operações de atualização referentes à associação CITY-STATUS. Deixamos ao leitor as considerações mais detalhadas sobre esses problemas. Comparando as Figs. 14.8 e 14.6, vemos que o efeito dessa reestruturação adicional foi o de eliminar a dependência transitiva de STATUS em S#.

Daremos agora uma definição da terceira forma normal.⁷

- Uma relação R está na *terceira forma normal* (3NF) se e somente se estiver na 2NF e todos os atributos não-chave forem dependentes não-transitivos da chave primária.

As relações SC e CS são ambas 3NF; a relação SECOND não é. (As chaves primárias de SC e CS são respectivamente S# e CITY.)⁹ Uma relação que esteja na segunda forma normal mas não na terceira pode sempre ser reduzida a uma coleção equivalente de relações 3NF. Já mencionamos que o processo é reversível, não havendo consequentemente perda de informação na redução; entretanto, a coleção 3NF pode conter informações, como o fato de o status de Roma ser 50, que não poderiam estar representadas na relação 2NF original. Assim como a estrutura SECOND/SP era uma representação um pouco melhor do mundo real do que a relação 1NF FIRST, a estrutura SC/CS é uma representação um pouco melhor da relação 2NF SECOND.

Vamos concluir esta seção repetindo que o nível de normalização de uma dada relação é uma questão de *semântica*, não dos valores dos dados que aparecem naquela relação em dado instante. Não é possível, pela simples observação de uma tabulação (exten-



Fig 14.8 Dependências funcionais nas relações SC e CS

SC	S=	CITY
S1	London	
S2	Paris	
S3	Paris	
S4	London	
S5	Athens	

CS	CITY	STATUS
Athens	30	
London	20	
Paris	10	

Fig. 14.9 Exemplos de tabulação de SC e CS.

7

Veja nota 3 de rodapé na página 242.

são) de determinada relação em dado momento, dizer-se se ela é ou não uma relação 3NF – é necessário conhecer-se o significado dos dados, isto é, as dependências envolvidas, antes de fazer tal julgamento. Em especial, o DBMS não pode garantir que uma relação seja mantida em 3NF (ou qualquer outra forma exceto 1NF) se não for informado de todas as dependências relevantes. Entretanto, para uma relação 3NF, tudo o que precisa ser informado ao DBMS sobre as dependências é uma indicação sobre o(s) atributo(s) constituinte(s) da chave primária. Assim o DBMS saberá que todos os outros atributos são funcionalmente dependentes deste atributo ou combinação de atributos, sendo capaz então de exigir essa restrição.⁸ Para uma relação que não esteja em 3NF, são necessárias especificações adicionais.

14.4 RELAÇÕES COM MAIS DE UMA CHAVE CANDIDATA

Como mencionamos na Seção 14.1, a definição de 3NF foi posteriormente substituída por uma mais completa. A nova definição deve-se a Boyce e Codd; por isso o termo “forma normal de Boyce/Codd (BCNF) é freqüentemente usado para distinguir esta nova 3NF da antiga. A definição da BCNF é conceitualmente mais simples do que a da 3NF, por não fazer referência explícita às primeira e segunda formas normais, nem aos conceitos de dependência total e transitiva. Chamemos a um atributo, possivelmente composto, do qual algum outro atributo é funcionalmente dependente, de *determinante* (funcional). Assim, podemos definir BCNF como se segue:

- Uma relação está na forma normal de Boyce/Codd (BCNF) se e somente se cada determinante for uma chave candidata.

Observe que estamos agora falando em termos de chaves *candidatas*, e não somente chave primária. A motivação para a introdução da BCNF é que a definição original de 3NF não manuseia satisfatoriamente o caso de uma relação que possua duas ou mais chaves candidatas compostas que se superpõem (veja abaixo).

Embora BCNF seja mais exigente (mais restritiva) do que 3NF, é ainda verdade que qualquer relação pode ser decomposta de uma forma sem perdas em uma coleção equivalente de relações BCNF.

Antes de vermos exemplos envolvendo mais de uma chave candidata, vamos entender que as relações FIRST e SECOND, que não eram 3NF pela definição antiga, também não são BCNF; e também que as relações SP, SC e CS, que eram 3NF sob a antiga definição, são também BCNF. A relação FIRST contém três determinantes: S#, CITY, e a combinação (S#, P#). Desses, somente (S#, P#) é uma chave candidata; consequentemente FIRST não é BCNF. De forma semelhante, SECOND não é BCNF, porque o determinante CITY não é uma chave candidata. Por outro lado, as relações SP, SC e CS são todas BCNF, pois em cada caso a chave primária é o único determinante na relação.

Vejamos agora um exemplo envolvendo duas chaves candidatas não-superpostas. Consideremos novamente a relação S(S#, STATUS, CITY). Vamos supor, como já fizemos antes neste livro, que STATUS e CITY são mutuamente independentes, e que também os *nomes* dos fornecedores, bem como seus números, sejam únicos (todo o tempo) – em outras palavras, estamos supondo que SNAME é uma chave candidata. Veja o diagrama de dependência da Fig. 14.3.

⁸ Veja nota de rodapé 3 na página 242.

A relação S é BCNF. No entanto, é desejável especificar-se *ambas* as chaves na definição da relação, (a) para informar ao DBMS, a fim de que este possa exigir as restrições devidas às dependências nos dois sentidos entre as duas chaves – ou seja, que em correspondência a um número de fornecedor, exista um único nome de fornecedor, e vice-versa; e (b) para informar aos usuários, pois naturalmente a unicidade dos dois atributos é um aspecto da semântica da relação, sendo consequentemente de interesse do pessoal que a utiliza. Veja a Fig. 4.8 (Capítulo 4).

Apresentaremos agora alguns exemplos nos quais as chaves candidatas se superpõem. Duas chaves candidatas se superpõem se cada uma delas envolver dois ou mais atributos e tiverem um atributo em comum. Como nosso primeiro exemplo, vamos novamente supor que os nomes dos fornecedores sejam únicos, e consideremos a relação SSP (S#, SNAME, P#, QTY). As chaves são (S#, P#) e (SNAME, P#). É esta relação BCNF? A resposta é não, pois temos dois determinantes, S# e SNAME, que não são chaves da relação (S# determina SNAME, e vice-versa). Mas a relação é 3NF pela antiga definição [14.1]; a definição original não exigia que um atributo fosse totalmente dependente da chave primária se fosse em si um componente de alguma outra chave da relação, e portanto o fato de SNAME não ser totalmente dependente de (S#, P#) – supondo que esta seja a chave primária – foi ignorado. Entretanto, este fato gera a redundância, e consequentemente problemas de atualização, na relação SSP. Por exemplo, a atualização do nome de Smith para Robinson nos leva (novamente) ou a problemas de pesquisa ou a resultados possivelmente inconsistentes. A solução para os problemas, como sempre, é a decomposição da relação SSP em duas projeções, neste caso SS(S#, SNAME) e SP(S#, P#, QTY) [ou SP(SNAME, P#, QTY)]. Estas projeções são ambas BCNF.

Como um segundo exemplo, consideremos a relação SJT com atributos S (estudante), J (assunto) e T (professor). O significado de uma tupla SJT é o de que um estudante específico recebe instrução sobre um assunto específico dada por um professor específico.

Seguem-se as regras de semântica:

- Para cada assunto, cada aluno daquele assunto recebe instrução de um só professor.
- Cada professor ministra somente um assunto.
- Cada assunto é ministrado por diversos professores.

A Fig. 14.10 mostra um exemplo de tabulação desta relação.

Quais são as dependências funcionais de SJT? Da primeira regra de semântica, temos uma dependência funcional de T no atributo composto (S, J). Da segunda regra de semântica, temos uma dependência funcional de J em T. A terceira regra de semântica nos informa que *não há* uma dependência funcional de T em J. Portanto nós temos a situação mostrada na Fig. 14.1.

SJT	S	J	T
	Smith	Math	Prof. White
	Smith	Physics	Prof. Green
	Jones	Math	Prof. White
	Jones	Physics	Prof. Brown

Fig. 14.10 Exemplo de tabulação da relação SJT.

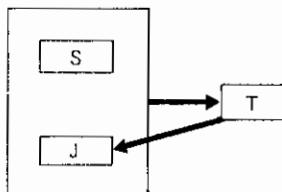


Fig. 14.11 Dependências funcionais na relação SJT.

De novo temos duas chaves candidatas se superpondo: a combinação (S, J) e a combinação (S, T). Outra vez a relação é 3NF mas não BCNF, e outra vez a relação padece de algumas anomalias em conexão com as operações de atualização. Por exemplo, se desejarmos remover a informação de que Jones estuda física, não poderemos fazê-lo sem também perder, ao mesmo tempo, a informação de que o professor Brown leciona física. As dificuldades são causadas pelo fato de T ser um determinante mas não uma chave candidata. Novamente poderemos superar o problema substituindo a relação original por duas projeções BCNF, neste caso ST(S, T) e TJ(T, J). Fica como um exercício para o leitor: fornecer as tabulações dessas duas relações com dados correspondentes aos da Fig. 14.10, desenhar o diagrama de dependência correspondente, garantir que as duas projeções são, sem dúvida, BCNF (quais são as chaves?),⁹ e verificar se esta solução realmente evita problemas. (Entretanto, ela introduz problemas diferentes. Veja a próxima seção.)

Nosso terceiro e último exemplo de chaves que se superpõem refere-se à relação EXAM com atributos S (estudante), J (assunto), e P (colocação). O significado de uma tupla EXAM é que um estudante específico foi examinado sobre um assunto específico e obteve uma colocação específica na turma. Para fins do exemplo, vamos supor que a seguinte regra de semântica prevaleça:

- Não há empates; isto é – não há dois estudantes que tenham obtido a mesma colocação no mesmo assunto.

Assim, as dependências funcionais são como ilustrado na Fig. 14.12.

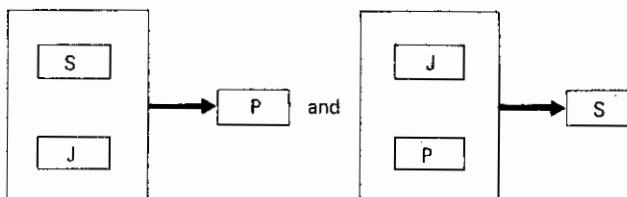


Fig. 14.12 Dependências funcionais na relação EXAM.

⁹ Na verdade, *qualquer* relação binária tem que estar na BCNF (por quê?)

De novo temos duas chaves candidatas, (S, J) e (J, P). A relação está BCNF embora as chaves se superponham, pois as chaves são os únicos determinantes. O leitor pode verificar que nesta relação não ocorrerão dificuldades nas operações de atualização como as discutidas anteriormente neste capítulo.

Uma declaração possível é:

```
RELATION EXAM(S, J, P)
PRIMARY KEY (S, J)
ALTERNATE KEY (J, P)
```

Vemos, portanto, que o conceito de BCNF elimina alguns casos de problemas que poderiam ocorrer sob a definição antiga de 3NF. Além disso, BCNF é conceitualmente mais simples do que 3NF, por não envolver referências aos conceitos de chave primária, dependência transitiva e dependência total. A referência a chaves candidatas podem também ser substituída por uma referência à noção mais fundamental de dependência funcional (a definição dada em [9.2] faz essa substituição). Por outro lado, os conceitos de dependência transitiva e total são úteis na prática, pois dão uma idéia do processo real por etapas que o projetista tem que percorrer para reduzir uma relação arbitrária a uma coleção equivalente de relações BCNF. Iremos summarizar este processo na Seção 14.8.

14.5 DECOMPOSIÇÕES BOAS E MÁS

Durante o processo de redução, ocorre freqüentemente que uma dada relação possa ser decomposta de várias maneiras. Considerando novamente a relação SECOND (S#, STATUS, CITY) dada na Seção 14.3 com as dependências funcionais (FDs)

$$\begin{aligned} \text{SECOND.S\#} &\rightarrow \text{SECOND.CITY} \\ \text{SECOND.CITY} &\rightarrow \text{SECOND.STATUS} \end{aligned}$$

e consequentemente também (por transitividade)

$$\text{SECOND.S\#} \rightarrow \text{SECOND.STATUS}$$

(veja a Fig. 14.13, na qual a FD transitiva está mostrada uma como uma seta tracejada). Mostramos na Fig. 14.3 que os problemas de atualização encontrados com SECOND poderiam ser superados substituindo-a pela sua decomposição nas duas projeções 3NF

$$\text{SC(S\#, CITY)} \quad \text{and} \quad \text{CS(CITY, STATUS)}$$

(de fato, SC e CS são não apenas 3NF, mas BCNF). Chamemos a esta decomposição de A. Como mencionado no início desta seção, a decomposição A não é a única possível. Uma decomposição alternativa é a B:

$$\text{SC(S\#, CITY)} \quad \text{and} \quad \text{SS(S\#, STATUS)}$$

(a projeção SC é a mesma tanto para A como para B). A decomposição B também não tem perdas, e novamente as duas projeções são BCNF. Mas a decomposição B é menos satisfatória do que a decomposição A, por diversas razões. Por exemplo, não é ainda possí-

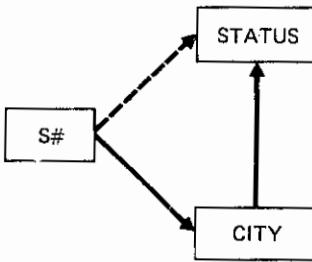


Fig. 14.13 Dependências funcionais na relação SECOND.

vel (em B) inserir-se o fato de que uma cidade específica tenha um determinado valor de status, a menos que haja algum fornecedor localizado naquela cidade.

Vamos examinar este exemplo mais a fundo. Primeiro, observe que as projeções na decomposição A correspondem às setas *cheias* na Fig. 14.13, enquanto que uma das projeções da decomposição B corresponde a uma seta *tracejada*. Na decomposição A, as duas projeções são *independentes* uma da outra, no sentido de que podem ser feitas atualizações a qualquer delas sem influência sobre a outra; desde que essa atualização seja legal, dentro do contexto da projeção envolvida — isto é, não viole a restrição FD que se aplica à projeção — então a *junção das duas projeções permanece sendo uma extensão legal da relação SECOND*. Isto é, esta junção não tem possibilidade de violar as restrições FD de SECOND. Em contraste, na decomposição B as atualizações a qualquer das projeções têm que ser monitoradas para garantir que não seja violada a FD SECOND.CITY → SECOND.STATUS. (Se dois fornecedores pertencerem à mesma cidade na projeção SC, eles terão que ter o mesmo status na projeção SS.) Portanto as projeções SC e SS *não* são independentes uma da outra.

O problema básico é que, na decomposição B, a FD CITY → STATUS tornou-se uma *restrição inter-relacional*. Por outro lado, na decomposição A, a restrição inter-relacional é a FD *transitiva* S# → STATUS, e esta restrição é automaticamente exigida se as duas restrições *intra-relacionais* S# → CITY e CITY → STATUS forem exigidas. (Para maior clareza, retiramos o prefixo “SECOND”.)

O conceito de projeções independentes fornece um guia para a escolha de uma decomposição específica quando há mais de uma possibilidade. Especificamente, uma decomposição na qual as projeções sejam independentes no sentido descrito é geralmente preferível a uma na qual elas não sejam. Rissanen [14.6] mostra que as projeções R1 e R2 de uma relação R são independentes nesse sentido se e somente se (a) cada FD em R puder ser logicamente deduzida daquelas em R1 e R2, e (b) pelo menos um par de atributos comuns de R1 e R2 formarem uma chave candidata. Este teorema torna a verificação de independência uma operação muito simples. Consideremos as decomposições A e B. Em A, as duas projeções são independentes, pois seu atributo comum CITY é a chave primária de CS, e cada FD em SECOND ou aparece em uma das duas projeções ou é uma consequência lógica daquelas que aparecem. Em contraste, em B as duas projeções não são independentes pois a FD CITY → STATUS não pode ser deduzida das FDs naqueles projeções (embora seu atributo comum S# seja a chave primária de ambas).

Como um aparte, observemos que a terceira possibilidade – substituição de SECOND por suas duas projeções

$SS(S#, STATUS)$ and $CS(CITY, STATUS)$,

não é uma decomposição válida, porque não é sem perdas. (*Exercício* – prove esta afirmativa.)

Uma relação que não pode ser decomposta em dois componentes independentes é dita ser *atômica* [14.6]. Não queremos dizer que uma relação não-atômica deva necessariamente ser decomposta em seus componentes atômicos; por exemplo as relações S e P do banco de dados de fornecedores e peças não são atômicas, mas tudo indica não ser importante decompô-las mais. (A relação SP, em contraste, é atômica.)

O fato de uma relação atômica não poder ser decomposta em componentes independentes não significa que não possa definitivamente ser decomposta. Consideremos a relação SJT da seção 14.4, com as FDs:

$$\begin{array}{l} (S, J) \rightarrow T \\ T \rightarrow J \end{array}$$

(vamos novamente ignorar os prefixos de nomes de relação). Como já vimos, esta relação (que é 3NF mas não BCNF) pode ser decomposta sem perdas em suas projeções ST(S, T) e TJ(T, J). Entretanto, essas projeções não são independentes, pelo teorema de Rissanen; a FD $(S, J \rightarrow T)$ não pode ser deduzida da FD $T \rightarrow J$ (a única FD representada nas duas projeções). Como resultado, as duas projeções não podem ser atualizadas independentemente (*exercício* – prove esta afirmativa); de fato, a relação SJT é atômica. Somos forçados à conclusão pouco agradável de que os objetivos de decompor uma relação em componentes BCNF e decompô-la em componentes independentes – podem ocasionalmente estar em conflito.

14.6 QUARTA FORMA NORMAL

Suponhamos que temos uma relação *não-normalizada* contendo informações sobre cursos, professores e textos. Cada registro da relação consiste de um nome de curso, mais um grupo repetitivo de nomes de professores, mais um grupo repetitivo de nomes de textos. A Fig. 14.14 mostra dois desses registros.

CTX	CURSO	PROFESSOR	TEXTO
	Physics	{ Prof. Green Prof. Brown Prof. Black }	{ Basic Mechanics Principles of Optics }
	Math	{ Prof. White }	{ Modern Algebra Projective Geometry }

Fig. 14.14 Exemplo de tabulação de CTX (não-normalizada).

O significado de um determinado registro nesta relação não-normalizada é que o curso especificado pode ser ministrado por qualquer dos professores indicados, e usa todos os textos indicados. Estamos supondo que, para um dado curso, possa existir qualquer quantidade de professores correspondentes e qualquer quantidade de textos correspondentes; além disso, vamos supor – talvez não muito realisticamente – que professores e livros são bastante independentes uns dos outros (isto é, não importa quem esteja no momento lecionando determinado curso, pois serão usados os mesmos textos). Vamos também supor que determinado professor ou texto possa ser associado com certa quantidade de cursos.

Vamos agora converter esta estrutura em uma forma normalizada equivalente. Observemos primeiro que não há quaisquer dependências funcionais nos dados (a não ser as triviais do tipo CURSO → CURSO). A teoria que desenvolvemos neste capítulo ainda não nos fornece uma base formal para decompor a estrutura em projeções; a única operação de natureza normalizante é a elementar de “aplanoar” a estrutura (como no Capítulo 4), o que para os dados da Fig. 14.14 produz a tabela mostrada na Fig. 14.15.

CTX	CURSO	PROFESSOR	TEXTO
	Physics	Prof. Green	Basic Mechanics
	Physics	Prof. Green	Principles of Optics
	Physics	Prof. Brown	Basic Mechanics
	Physics	Prof. Brown	Principles of Optics
	Physics	Prof. Black	Basic Mechanics
	Physics	Prof. Black	Principles of Optics
	Math	Prof. White	Modern Algebra
	Math	Prof. White	Projective Geometry

Fig. 14.15 Exemplo de tabulação de CTX (normalizada).

O significado da relação normalizada CTX é: uma tupla $\langle c, t, x \rangle$ aparece em CTX se e somente se o curso c puder ser ministrado pelo professor t usando o texto x como referência. Observe que, para um determinado curso, aparecem todas as combinações possíveis de professor e texto – isto é, CTX satisfaz à restrição:

se aparecerem ambas as tuplas $\langle c, t_1, x_1 \rangle, \langle c, t_2, x_2 \rangle$

então aparecerão também as tuplas $\langle c, t_1, x_2 \rangle, \langle c, t_2, x_1 \rangle$

10

O leitor poderá objetar que não é necessário incluir-se todas as combinações professor/texto de um dado curso; por exemplo, três tuplas são suficientes para mostrar que o curso de física tem três professores e dois textos. O problema é – *quais* três? Qualquer escolha particular leva a uma relação tendo uma interpretação não-óbvia e um comportamento muito estranho na atualização. Para ver que isto é verdade, o leitor deve tentar a experiência de fazer uma escolha e depois estabelecer o significado da relação resultante – isto é, estabelecer o critério para decidir se uma determinada tupla é ou não aceitável como membro daquela relação.

É evidente que a relação CTX contém uma boa quantidade de redundâncias, que causarão os problemas usuais nas operações de atualização. Por exemplo, para adicionar-se a informação de que o curso de física passou a usar um novo texto chamado *Advanced Mechanics*, torna-se necessário criar *três* novas tuplas – uma para cada um dos três professores.¹⁰ Ainda assim, CTX está na BCNF, pois ela é “toda chave” e não há outros determinantes funcionais. A existência dessas relações BCNF problemáticas foi reconhecida há algum tempo; veja, por exemplo, Schmidt and Swenson [14.23]. No que toca à relação CTX, fica intuitivamente claro que as dificuldades são causadas pelo fato de professores e textos serem independentes um do outro; é fácil também ver-se que haveria uma melhora se CTX fosse substituída por suas duas projeções CT(CURSO, PROFESSOR) e CX(CURSO, TEXTO). Veja a Fig. 14.16 (CT e CX são ambas “todas chaves”, sendo portanto BCNF).

CT	CURSO	PROFESSOR	CX	CURSO	TEXTO
	Physics	Prof. Green		Physics	Basic Mechanics
	Physics	Prof. Brown		Physics	Principles of Optics
	Physics	Prof. Black		Math	Modern Algebra
	Math	Prof. White		Math	Projective Geometry

Fig. 14.16 Exemplos de tabulação de CT e CX.

Entretanto, já dissemos que a decomposição da Fig. 14.16 não pode ser feita na base de dependências funcionais. Ao invés, ela é feita na base de um novo tipo de dependência, a de *múltiplos valores*. As dependências de múltiplos valores (MVDs) são uma generalização das dependências funcionais (isto é, uma FD é um caso especial de uma MVD).

Há duas MVDs na relação CTX:

$$\begin{array}{l} \text{CTX.COURSE} \rightarrow \rightarrow \text{CTX.TEACHER} \\ \text{CTX.COURSE} \rightarrow \rightarrow \text{CTX.TEXT} \end{array}$$

(A afirmativa MVD “R.A → → R.B é lida como “o atributo R.B é *multidependente* do atributo R.A”, ou, equivalentemente, “o atributo R.A *multidetermina* o atributo R.B”). Por enquanto vamos nos concentrar na primeira MVD acima, que intuitivamente significa que, embora um curso não tenha um *único* professor correspondente (isto é, PROFESSOR não é funcionalmente dependente de CURSO), cada curso tem um *conjunto* bem definido de professores correspondentes. Por “bem definido” queremos dizer mais precisamente que, para um curso *c* e um texto *x*, o conjunto *t* de professores que corresponde ao par (*c*, *x*) em CTX depende só de *c* – não faz diferença que valor de *x* escolhamos, desde que *c* e *x* apareçam juntas em alguma tupla de CTX. A segunda MVD (de TEXTO em CURSO) é interpretada de maneira análoga.

Daremos agora uma definição de MVD.

- Dada uma relação R com atributos A, B, e C, a *dependência de múltiplos valores*

$$R.A \rightarrow \rightarrow R.B$$

Vale para R, se e somente se o conjunto de valores B que se combinam com um dado par (valores de A, valores de C) em R depender somente do valor de A e for independente do valor de C. Como sempre, A, B, e C podem ser compostos.

Observe que as MVDs que definimos só podem existir se a relação R tiver pelo menos três atributos.¹¹

É fácil mostrar-se (veja [14.4]) que, dada a relação R(A, B, C), a MVD $R.A \rightarrow\rightarrow R$. B vale se e somente se a MVD $R.A \rightarrow\rightarrow R.C$ também valer. As MVDs estão sempre aos pares nesta forma. Por esta razão, é comum expressar-se ambas em uma mesma instrução, usando a notação

$R.A \rightarrow\rightarrow R.B \mid R.C$

Por exemplo,

C U R S O $\rightarrow\rightarrow$ P R O F E S S O R | T E X T O

(retirados os prefixos de nomes de relações).

A FD é uma MVD na qual o “conjunto” de valores dependentes consiste de um único. Voltando ao nosso problema de normalização, podemos verificar que a dificuldade com relações como a CTX é que elas envolvem MVDs que não são também FDs. As duas projeções CT e CX não envolvem nenhuma dessas MVDs, razão pela qual elas representam uma melhoria sobre a relação original. Gostaríamos portanto de substituir CTX por essas duas projeções. Há um teorema provado por Fagin [14.4] que nos permite efetuar essa substituição.

- Uma relação R, com atributos A, B, e C, pode ser decomposta sem perdas em suas duas projeções $R1(A, B)$ e $R2(A, C)$ se e somente se a MVD $A \rightarrow\rightarrow B \mid C$ valer em R. [Esta é uma versão mais potente do teorema de Heath (veja Seção 14.3).
Podemos agora definir a *quarta forma normal* (4NF).]
- Uma relação R está na quarta forma normal (4NF) se e somente se, sempre que existir uma MVD em R, digamos $A \rightarrow\rightarrow B$, todos os atributos de R sejam *funcionalmente* dependentes de A (isto é, $A \rightarrow X$ para todos os atributos X de R).

Em outras palavras, as únicas dependências (FDs ou MVDs) em R são da forma $K \rightarrow X$ (isto é, uma dependência funcional de uma chave candidata K em algum outro atributo X).

Podemos agora verificar que a relação CTX não está na 4NF, pois ela envolve uma MVD que definitivamente não é uma FD, não mencionando uma FD na qual o determinante é uma chave candidata. No entanto, as duas projeções CT e CX estão na 4NF. Por isso, 4NF representa um avanço sobre a BCNF ao eliminar outra forma de estrutura indesejável.

¹¹ A definição dada em [14.4] não requer que a relação R tenha pelo menos três atributos. Por simplicidade, estamos ignorando alguns casos especiais; por exemplo, a MVD “trivial” $R.X \rightarrow\rightarrow R.Y$ que sempre vale na relação binária $R(X, Y)$ e algumas MVDs não-triviais nas quais o lado esquerdo é o conjunto vazio. Veja [14.4] para detalhes; veja também o Exercício 14.5.

Fagin provou dois outros resultados importantes em [14.4], que nos permitem incorporar a 4NF no procedimento global de normalização desenvolvido gradualmente neste capítulo:

1. A 4NF é estritamente mais potente do que a BCNF — isto é, qualquer relação 4NF está necessariamente na BCNF.
2. Qualquer relação pode ser decomposta sem perdas em uma coleção equivalente de relações 4NF.

Em outras palavras, sempre se pode chegar à 4NF — embora os resultados da Seção 14.5 mostrem que pode não ser desejável em alguns casos levar tão longe a decomposição (nem mesmo até a BCNF). É importante observar, incidentalmente, que o trabalho de Rissanen sobre projeções independentes [14.6], embora baseado em termos de FDs, é também aplicável a MVDs. Lembre-se de que uma relação $R(A, B, C)$ satisfazendo às FDs $A \rightarrow B$, $B \rightarrow C$ é melhor decomposta em suas projeções sobre (A, B) e (B, C) do que sobre as (A, B) , (A, C) . O mesmo se aplica quando as FDs são substituídas pelas MVDs $A \rightarrow \rightarrow B$, $B \rightarrow \rightarrow C$.

14.7 QUINTA FORMA NORMAL

Até agora neste capítulo estamos supondo tacitamente que a única operação necessária ou disponível no processo de decomposição é a substituição de uma relação por duas de suas projeções. Esta posição nos levou com êxito até a 4NF. Talvez cause surpresa descobrirmos que existem relações que não podem ser decompostas sem perdas em duas projeções, mas *podem* ser decompostas sem perdas em três (ou mais). Este fenômeno foi primeiramente observado por Aho, Beeri, e Ullman [12.16], tendo sido também estudado por Nicolas [14.9]. Consideremos a relação SPJ (Fig. 14.17). Esta relação é “toda chave” e não envolve FDs e MVDs não-triviais, estando então na 4NF. A Fig. 14.17 também mostra (a) as três projeções SP, PJ, e JS de SPJ, e (b) o efeito da junção de SP e PJ sobre P# e depois a junção do resultado e JS sobre (J#, S#). Observe que o resultado da primeira junção é produzir uma cópia da SPJ original mais uma tupla espúria, e que o efeito da segunda junção é eliminar aquela tupla. (O resultado final é o mesmo, qualquer que seja o par de projeções que escolhamos para a primeira junção, embora o resultado intermediário seja diferente em cada caso.)

O exemplo da Fig. 14.17 está naturalmente expresso em termos de *extensões*. Entretanto, a “decompossibilidade em três” (usando um termo feio porém convincente) de SPJ poderá ser uma propriedade *intensional* mais fundamental — isto é, uma propriedade satisfeita por todas as extensões legais — se a relação satisfizer a uma restrição independente do tempo. Para entendermos que restrição tem que ser essa, observemos primeiro que a afirmativa de que SPJ é igual à junção¹² de suas três projeções SP, PJ, e JS é equivalente à afirmativa:

¹² Aho, Beeri, e Ullman [12.16] fornecem uma definição genérica de “junção” que nos permite falar sem ambigüidade sobre junção de qualquer quantidade de relações. A junção genérica de uma lista ordenada de relações é formada pela repetição de substituições da primeira e segunda relações da lista pela sua junção natural, até que toda a lista tenha sido reduzida a uma única relação.

se o par $\langle s_1, p_1 \rangle$ aparece em SP
 e o par $\langle p_1, j_1 \rangle$ aparece em PJ
 e o par $\langle j_1, s_1 \rangle$ aparece em JS
 então a tripla $\langle s_1, p_1, j_1 \rangle$ aparece em SPJ

(porque a tripla $\langle s_1, p_1, j_1 \rangle$ obviamente aparece na junção de SP, PJ, e JS)¹³. Uma vez que $\langle s_1, p_1 \rangle$ aparece em SP se e somente se s_1 e p_1 aparecerem juntos em SPJ, o mesmo acontecendo com $\langle p_1, j_1 \rangle$ e $\langle j_1, s_1 \rangle$, podemos reescrever a última afirmativa como uma restrição sobre SPJ:

se $\langle s_1, p_1, j_2 \rangle, \langle s_2, p_1, j_1 \rangle, \langle s_1, p_2, j_1 \rangle$ aparecem em SPJ
 então $\langle s_1, p_1, j_1 \rangle$ também aparecem em SPJ.

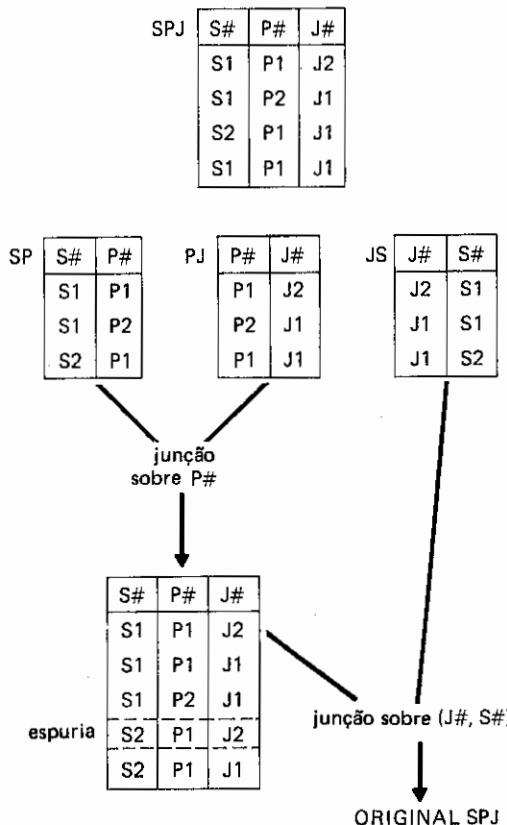


Fig. 14.17 SPJ é a junção de três de suas projeções, mas não de quaisquer duas.

¹³ O reverso desta afirmativa, isto é, se $\langle s_1, p_1, j_1 \rangle$ aparecem em SPJ então $\langle s_1, p_1 \rangle$ aparece na sua projeção SP (etc.), é obviamente verdadeiro para qualquer relação SPJ de grau 3.

Isto é uma restrição (embora bastante bizarra), tal como uma FD em uma MVD. Como isto é satisfeito se e somente se a relação envolvida for a junção de certas projeções suas, esta restrição é denominada *dependência de junção* (JD). No exemplo, dizemos que SPJ satisfaz a dependência de junção “ * (SP, PJ, JS)”. Em geral, a relação satisfaz à JD * (X, Y, ..., Z) se e somente se ela for a junção de suas projeções em X, Y, ..., Z, onde X, Y, ..., Z são subconjuntos do conjunto de atributos de R.

Já vimos que a relação SPJ, com a sua dependência de junção * (SP, PJ, JS), pode ser “decomposta em três”. A pergunta é: *deve ser?* A resposta será provavelmente sim. A relação SPJ padece de uma série de problemas nas operações de atualização, que são removidos quando ela é “decomposta em três”. Alguns exemplos estão mostrados na Fig. 14.18. Deixamos como exercício para o leitor as considerações sobre o que acontece após a “*decomposição em três*”.

O teorema de Fagin (Seção 14.6) de que R(A, B, C) pode ser decomposta sem perdas em R1 (A, B) e R2 (A, C) se e somente se $A \rightarrow\rightarrow B$ valer em R, é equivalente à afirmativa de que R(A, B, C) satisfaz à JD * (AB, AC) se e somente se satisfizer à MVD $A \rightarrow\rightarrow B | C$. Como o teorema pode ser usado como uma *definição* de MVD, segue-se que uma MVD é um caso especial de uma JD, ou que JDs são uma generalização das MVDs (assim como as MVDs são uma generalização das FDs). Além disso, é imediato da definição que as JDs são a forma *mais* geral de dependência possível — isto é, não existe uma forma ainda mais alta de dependência da qual a JD seja por sua vez um mero caso especial — na medida em que nossa atenção fique restrita a dependências que lidam com uma relação sendo decomposta via projeção e recomposta via junção. (Entretanto se permitirmos tipos adicionais de operadores no processo de decomposição e recomposição, surgirão tipos adicionais de dependências. Vamos discutir isto brevemente no final deste capítulo.)

Voltando ao nosso exemplo, o problema com a relação SPJ é que, embora sendo 4NF, ela ainda envolve uma JD (que não é nem FD nem MVD). Vimos, portanto, que é possível, e talvez desejável, decompor tal relação em componentes menores — ou seja, nas projeções especificadas pela dependência de junção. O processo de decomposição pode ser repetido até que todas as projeções estejam na *quinta forma normal* (5NF).

- Uma relação R está na quinta forma normal (5NF) — também chamada de forma normal projeção-junção (PJ/NF) — se e somente se cada dependência de junção em

SPJ	S#	P#	J#
S1	P1	J2	
S1	P2	J1	

- se $\langle S2, P1, J1 \rangle$ inseridas,
 $\langle S1, P1, J1 \rangle$ também têm
que ser inseridas

- o inverso não é verdadeiro

SPJ	S#	P#	J#
S1	P1	J2	
S1	P2	J1	
S2	P1	J1	
S1	P1	J1	

- pode remover $\langle S2, P1, J1 \rangle$
sem efeitos colaterais

- se $\langle S1, P1, J1 \rangle$ removida,
outra tupla tem também que
ser removida (qual?)

Fig. 14.18 Exemplos de problemas de atualização em SPJ.

R estiver subentendida pelas chaves candidatas de R. (Vamos ampliar adiante a noção de uma JD “subentendida por chaves candidatas”.)

A relação SPJ não é 5NF; sua única chave candidata, a combinação (S#, P#, J#), certamente não subentende que a relação possa ser decomposta sem perdas em suas projeções SP, PJ e JS. As projeções SP, PJ, e JS estão na 5NF, pois não envolvem qualquer JD.

Observemos que, sendo a MVD um caso especial da JD, qualquer relação 5NF está automaticamente também na 4NF. (Fagin mostra em [14.5] que uma MVD subentendida por uma chave candidata tem que ser de fato uma FD, na qual a chave é o determinante.) Como mencionado anteriormente, qualquer relação pode ser decomposta sem perdas em uma coleção equivalente de relações 5NF.

Voltando à questão da JD ser “subentendida” por chaves, vamos considerar um exemplo simples. A relação de fornecedores S(S#, SNAME, STATUS, CITY), cujas chaves candidatas são S# e SNAME, satisfaz a diversas dependências de junção; por exemplo a JD:

* ((S#, SNAME, STATUS), (S#, CITY))

Isto é, a relação S é igual à junção de suas projeções em (S#, SNAME, STATUS) e (S#, CITY). Esta JD está subentendida por ser S# uma chave candidata (pelo teorema de Heath). A relação S também satisfaz à JD:

* ((S#, SNAME), (S#, STATUS), (SNAME, CITY))

Esta JD está subentendida por serem *ambas* S# e SNAME chaves candidatas. Fagin [14.5] fornece um algoritmo pelo qual é possível, dada uma JD e um conjunto de chaves candidatas, testar se a JD está subentendida por aquelas chaves (em geral, não é imediatamente óbvio—como prova o segundo exemplo acima). Assim, dada uma relação R, podemos dizer se R está na 5NF se conhecermos as chaves candidatas e *todas as JDs* em R. No entanto, a descoberta das JDs em si não é uma operação trivial, isto é: enquanto a descoberta das FDs e MVDs é relativamente fácil (porque elas têm uma interpretação bastante direta no mundo real), o mesmo não pode ser dito para uma JD que não seja uma MVD (porque o significado intuitivo dessa JD está longe de ser direto). Consequentemente, o processo de se determinar quando uma relação está na 4NF mas não na 5NF (e por isso poderia ser vantajosamente decomposta) não é claro. É atraente mencionarmos que essas relações são casos patológicos e de rara probabilidade de ocorrência na prática.

Concluindo, observamos que, por definição, a 5NF é a *última* forma normal relativa a projeção e junção, pois, se uma relação estiver na 5NF, as únicas decomposições válidas são as baseadas nas chaves candidatas (de tal forma que cada projeção consiste de uma ou mais chaves candidatas mais zero ou mais atributos). Por exemplo, a relação S de fornecedores está na 5NF. Ela *pode* ser decomposta sem perdas de várias maneiras, como vimos acima, mas cada projeção ainda conterá pelo menos uma das duas chaves, não nos parecendo por isso haver qualquer vantagem nessas decomposições.

14.8 RESUMO

Este capítulo esteve voltado para o uso da técnica de *decomposição sem perdas* como um auxílio ao projeto de bancos de dados. A idéia básica é a de que começamos com alguma

relação, juntamente com o estabelecimento de certas restrições (FDs, MVDs, JDs), e sistematicamente reduzimos a relação a uma coleção de relações que são equivalentes à original embora numa forma preferível a esta, usando as restrições como guia durante o processo de redução. Podemos resumir informalmente o processo de redução como se segue:

- a) Obtenha projeções da relação 1NF original para eliminar qualquer dependência funcional não-total. Isto produzirá uma coleção de relações 2NF.
- b) Obtenha projeções dessas relações 2NF para eliminar qualquer dependência transitiva. Isto produzirá uma coleção de relações 3NF.
- c) Obtenha projeções dessas relações 3NF para eliminar qualquer dependência funcional restante na qual o determinante não seja uma chave candidata. Isto produzirá uma coleção de relações BCNF. [Nota: as etapas (a) – (c) podem ser condensadas em uma única orientação – “Obtenha projeções da relação original para eliminar as FDs nas quais o determinante não seja uma chave candidata”.]
- d) Obtenha projeções dessas relações BCNF para eliminar qualquer dependência de múltiplos valores que não seja também dependência funcional. Isto produzirá uma coleção de relações 4NF. (Nota: na prática é usual eliminar-se essas MVDs *antes* da aplicação das outras etapas acima.)
- e) Obtenha projeções dessas relações 4NF para eliminar qualquer dependência de junção que não esteja subentendida pelas chaves candidatas (talvez devamos acrescentar “se você puder achá-las”).

Em cada etapa do processo o conceito de componentes independentes (Seção 14.5) pode ser usado como guia para a escolha de que projeções obter.

O objetivo geral do processo de redução é o de reduzir a redundância, evitando consequentemente alguns problemas nas operações de atualização. Mas deve ser novamente enfatizado que as orientações sobre normalização são somente orientações; há às vezes boas razões para não se normalizar “até o fim” (embora o projetista deva documentar e justificar desvios daquela posição extrema). O exemplo clássico é o da relação de nomes e endereços:

NOMEND (NOME, RUA, CIDADE, ESTADO, CEP)

na qual estamos supondo que, em adição às FDs subentendidas por NOME (a chave primária), temos também a FD

CEP D (CIDADE, ESTADO)

Esta relação certamente não está na 5NF (em que forma está?), e o processo de redução esboçado acima nos indica que a decomponhamos nas projeções em (NOME, RUA, CEP) e (CEP, CIDADE, ESTADO). No entanto, como os códigos de endereçamento postal não variam com frequência, e RUA, CIDADE, e ESTADO são invariavelmente necessários juntos, não parece que esta decomposição seja conveniente. (De fato, a estrutura de dependência da relação NOMEND é ainda pior do que a mencionada, pois também envolve a FD $(\text{RUA}, \text{CIDADE}, \text{ESTADO}) \rightarrow \text{CEP}$.)

Podemos observar que o tópico deste capítulo é de espécie diferente do de alguns capítulos anteriores. As noções de dependência e normalização são voltadas ao *significado* dos dados. Em contraste, linguagens do tipo álgebra relacional e cálculo relacional são vol-

tadas somente para os valores atuais dos dados, sendo qualquer interpretação daqueles valores imposta de fora, não fazendo parte da linguagem em si. Em particular, essas linguagens não exigem que as relações sobre as quais elas operam estejam em alguma forma específica diferente de 1NF. A normalização pode ser encarada basicamente como uma *disciplina* – uma disciplina através da qual o projetista pode captar uma parte, embora pequena, do mundo real da empresa que o banco de dados representa.

Vamos concluir este capítulo voltando à observação feita na Seção 14.1 sobre não existirem outras formas normais além das mostradas na Fig. 14.2. Vamos discutir brevemente aqui duas formas normais adicionais, para dar uma idéia sobre como está prosseguindo a pesquisa sobre normalização. Primeiro, Smith [14.25] examinou a possibilidade de reduzir uma relação dividindo-a “horizontalmente” (ao invés de “verticalmente”, como a projeção faz), de forma que a relação original possa ser recuperada via uma operação de união; essas considerações o levaram a definir uma forma normal que ele chama de (3,3)NF. (3,3)NF subentende BCNF; entretanto, uma relação (3,3)NF não tem que estar na 4NF, nem uma relação 4NF precisa ser (3,3)NF, de tal forma que uma redução a (3,3)NF é “ortogonal” à redução 4NF e 5NF. Segundo, Fagin [14.26] definiu a “forma normal domínio-chave” (DK/NF), na qual as noções de FD, MVD e JD não estão mencionadas. Uma relação está na DK/NF se e somente se cada restrição na relação for uma consequência lógica das *restrições de chaves* e *restrições de domínios*. (Restrição de chave é o estabelecimento de que determinado atributo ou combinação de atributos é uma chave candidata; restrição de domínio é o estabelecimento de que os valores de determinado atributo se encontram dentro de algum conjunto prescrito de valores.) É portanto conceitualmente simples a exigência das restrições em uma relação DK/NF. Fagin também mostra que qualquer relação DK/NF é automaticamente 5NF (e consequentemente 4NF etc.), e sem dúvida também (3,3)NF. Entretanto, DK/NF nem sempre é alcançável, nem foi respondida a pergunta “exatamente quando *pode* ela ser conseguida?”.

EXERCÍCIOS

14.1 A Fig. 14.19 representa uma estrutura hierárquica (veja o Capítulo 3) que contém informações sobre departamentos de uma companhia. Para cada departamento, o banco de dados contém um número de departamento (único), um valor de verba e o número (único) de empregado do respectivo gerente. Para cada departamento, o banco de dados também contém informações sobre todos os empregados que trabalham nesse departamento, todos os projetos a ele designados e todos os escri-

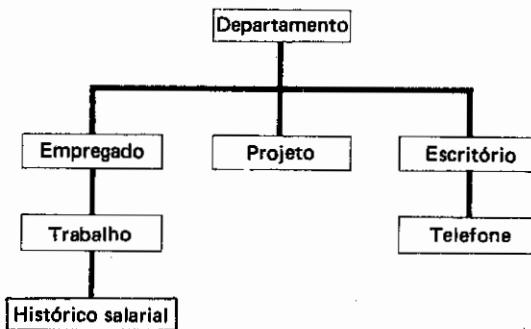


Fig. 14.19 Banco de dados de uma companhia (estrutura hierárquica).

tórios ocupados por esse departamento. As informações sobre empregados consistem do número (único) de cada empregado, o número do projeto no qual ele trabalha, o número e o telefone do escritório; as informações sobre projetos consistem do número (único) e um valor de verba; e as informações sobre escritórios consistem de um número (único) de escritório e a sua área (em metros quadrados). Além disso, o banco de dados contém para cada empregado os títulos dos trabalhos que este já realizou, juntamente com a data e o salário para cada salário diferente que ele recebeu em cada trabalho; e para cada escritório, contém as números (únicos) de todos os telefones daquele escritório.

Converta esta estrutura hierárquica em uma coleção apropriada de relações normalizadas. Faça quaisquer suposições que achar razoáveis sobre as dependências envolvidas.

14.2 Um banco de dados usado em um sistema de entrada de dados deve conter informações sobre *clientes*, *itens* e *pedidos*. As seguintes informações devem estar incluídas:

- Para cada *cliente*:
 - Número do cliente (único)
 - Endereço válido para remessa (diversos por cliente)
 - Saldo
 - Limite de crédito
 - Desconto
- Para cada *pedido*:
 - Cabeçalho informativo: número do cliente, endereço para remessa, data do pedido
 - Linhas de detalhe (diversas por pedido), cada uma dando o número do item e a quantidade pedida.
- Para cada *item*:
 - Número do item (único)
 - Fábricas produtoras
 - Quantidade estocada em cada fábrica
 - Nível de risco do estoque de cada fábrica
 - Descrição do item

Por motivos internos de processamento, o valor “quantidade a ser remetida” fica associado a cada linha de detalhe de cada pedido. [Este valor é inicialmente colocado como igual ao pedido para o item e (progressivamente) reduzido até zero à medida que as remessas (parciais) são feitas.]

Projete o banco de dados para estes dados. Como na questão anterior, faça as suposições que julgar necessárias.

14.3 Suponhamos que no Exercício 14.2 somente uma quantidade muito pequena de clientes, digamos um por cento, tenha mais do que um endereço para remessa. (Isto é típico nas situações da vida real, onde é muito frequente que umas poucas exceções – usualmente bastante importantes – deixem de seguir a regra geral.) Isto cria algum empecilho na sua relação do Exercício 14.2? É possível melhorá-la?

14.4 Uma relação HORÁRIO está definida com os seguintes atributos:

- D Dia da semana (1–5)
- P Período durante o dia (1–8)
- C Nº da sala de aula
- T Nome do professor
- S Nome do estudante
- L Identificador da aula

Uma tupla (d, p, c, t, s, l) é um elemento desta relação se no momento (d, p) o estudante s participa da aula l ministrada pelo professor t na sala c . Podemos supor que todas as aulas têm a duração de um período e que cada aula possui um identificador que é único entre todas as aulas ministradas na semana. Reduza HORÁRIO a uma estrutura mais desejável.

14.5 Qual das seguintes afirmativas é verdadeira? Para as que não forem, prepare outro exemplo que se encaixe.

- a) Qualquer relação binária está na 3NF.
- b) Qualquer relação binária está na BCNF.
- c) Qualquer relação binária está na 4NF.

- d) Qualquer relação binária está na PJ/NF.
- e) Uma relação R(A, B, C) é igual à junção de suas projeções R1(A, B) e R2(A, C) se e somente se a FD A → B valer em R.
- f) Se R.A → R.B e R.B → R.C, então R.A → R.C
- g) Se R.A → R.B e R.A → R.C, então R.A → R.(B, C)
- h) Se R.B → R.A e R.C → R.A, então R.(B, C) → R.A

14.6 Um banco de dados deve conter informações sobre representantes de vendas, vendas, áreas e produtos. Cada representante é responsável pelas vendas em uma ou mais áreas; cada área tem um ou mais representantes responsáveis. Semelhantemente, cada representante é responsável pela venda de um ou mais produtos, e cada produto tem um ou mais representantes responsáveis. Todos os produtos são vendidos em todas as áreas; no entanto, não há dois representantes vendendo o mesmo produto na mesma área. Cada representante vende o mesmo conjunto de produtos em cada área onde atua como responsável. Projete uma estrutura adequada para estes dados.

REFERÊNCIAS E BIBLIOGRAFIA

- 14.1 E. F. Codd. "Further Normalization of the Data Base Relational Model." In *Data Base Systems*, Courant Computer Symposia Series, Vol. 6. Englewood Cliffs, N. J.: Prentice-Hall (1972).
- 14.2 E. F. Codd. "Normalized Data Base Structure: A Brief Tutorial." Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.
- 14.3 I. J. Heath. "Unacceptable File Operations in a Relational Database." Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.
- 14.4 R. Fagin. "Multivalued Dependencies and a New Normal Form for Relational Databases". *ACM Transactions on Database Systems* 2, nº 3 (setembro de 1977).
- 14.5 R. Fagin. "Normal Forms and Relational Database Operators." Proc. 1979 ACM SIGMOD International Conference on Management of Data.
- 14.6 J. Rissanen. "Independent Components of Relations." *ACM Transactions on Database Systems* 2, nº 4 (dezembro de 1977).
- 14.7 W. W. Armstrong. "Dependency Structures of Data Base Relationships." Proc. IFIP Congress 1974.

O artigo que primeiro formalizou a teoria das FDs. Esta teoria fornece um conjunto de axiomas que caracterizam com precisão todas as possíveis estruturas de FDs dentro de uma relação. Os axiomas podem ser estabelecidos como se segue.

1. $A_1 A_2 \cdots A_m \rightarrow A_i$, para $i = 1, 2, \dots, m$.
2. $A_1 A_2 \cdots A_m \rightarrow B_1 B_2 \cdots B_r$
se e somente se
 $A_1 A_2 \cdots A_m \rightarrow B_i$, para cada i em $1, 2, \dots, r$.
3. se $A_1 A_2 \cdots A_m \rightarrow B_1 B_2 \cdots B_r$
e $B_1 B_2 \cdots B_r \rightarrow C_1 C_2 \cdots C_p$,
então $A_1 A_2 \cdots A_m \rightarrow C_1 C_2 \cdots C_p$.

Estes axiomas são *completos* no sentido de que, dado um conjunto S de FDs, todas as FDs subentendidas por S podem ser derivadas de S usando-se os axiomas, e *precisos*, no sentido de que não podem ser derivadas FDs adicionais (isto é, FDs não subentendidas por S). O artigo fornece também uma caracterização completa das chaves candidatas.

- 14.8 C. Beeri, R. Fagin, and J. H. Howard. "A Complete Axiomatization for Functional and Multivalued Dependencies." Proc. 1977 ACM SIGMOD International Conference on Management of Data. Estende o trabalho de Armstrong [14.7] para incluir tanto MVDs como FDs.
- 14.9 J. M. Nicolas. "Mutual Dependencies and Some Results on Undecomposable Relations." Proc. 4th International Conference on Very Large Data Bases (1978).

Introduz o conceito de "dependência mútua" (na verdade um caso especial da dependência de

junção mais geral – isto é, uma JD que não é MVD ou FD – envolvendo exatamente três projeções). Esta noção nada tem a ver com o conceito de dependência mútua mencionada na Seção 14.3.

- 14.10 J. Rissanen and C. Delobel. "Decomposition of Files, a Basis for Data Storage and Retrieval." IBM Research Report RJ 1220 (maio de 1973).

Um dos tratamentos formais mais antigos.

- 14.11 R. G. Casey and C. Delobel. "Decomposition of a Data Base and the Theory of Boolean Switching Functions." *IBM J. R & D* 17, nº 5 (setembro de 1973).

Mostra que para qualquer relação 1NF dada o conjunto de dependências funcionais (chamadas de relações funcionais neste artigo) pode ser representado por uma função Booleana, e que também esta função é única no seguinte sentido: as dependências originais podem ser especificadas de várias maneiras diferentes (superficiais), cada uma geralmente dando origem a funções Booleanas (superficialmente) diferentes – mas todas essas funções podem ser reduzidas pelas leis da álgebra Booleana à mesma forma canônica. O problema da decomposição de uma relação 1NF é então mostrado como sendo logicamente equivalente ao bem conhecido problema da álgebra Booleana de encontrar-se um conjunto de implicants primos da função Booleana correspondente à relação original juntamente com suas dependências funcionais. Conseqüentemente o problema pode ser transformado em um problema da álgebra Booleana, que tem várias vantagens potenciais. Por exemplo, torna-se possível usar-se técnicas analíticas de modelagem para avaliar decomposições alternativas.

- 14.12 C. Delobel and D. S. Parker. "Functional and Multivalued Dependencies in Relational Database and the Theory of Boolean Switching Functions." Tech. Report nº 142, Dept. Maths. Appl. et Informatique, Univ. de Grenoble, France (novembro de 1978).

Estende os resultados de [14.11] para incorporar as MVDs.

- 14.13 D. S. Parker and C. Delobel. "Algorithmic Applications for a New Result on Multivalued Dependencies." *Proc. 5th International Conference on Very Large Data Bases* (1979).

Aplica os resultados de [14.12] a diversos problemas, como o de testar se uma decomposição é sem perdas.

- 14.14 R. Fagin. "Functional Dependencies in a Relational Database and Propositional Logic." *IBM J. R & D.* 21, nº 6 (novembro de 1977).

Mostra que os axiomas de Fagin [14.7] são estritamente equivalentes ao sistema de instruções implicativas na lógica proposicional; isto é, uma dada instrução de dependência é conseqüência de um dado conjunto de instruções de dependências se e somente se a instrução implicativa correspondente for conseqüência do conjunto correspondente de instruções implicativas.

- 14.15 Y. Sagiv and R. Fagin. "An Equivalence Between Relational Database Dependencies and a Subclass of Propositional Logic." IBM Research Report RJ2500 (março de 1979).

Estende os resultados de [14.14] para incluir MVDs.

- 14.16 P. A. Bernstein. "Synthesizing Third Normal Form Relations from Functional Dependencies." *ACM Transactions on Database Systems* 1, nº 4 (dezembro de 1976).

Este capítulo apresentou a abordagem de *decomposição* para o problema de projeto. Bernstein propõe a abordagem inversa: dado um conjunto de atributos e um conjunto de dependências funcionais sobre eles, deve ser possível *sintetizar*-se um conjunto apropriado de relações. São apresentados algoritmos para executar esta tarefa. No entanto, como os atributos (e portanto as dependências funcionais) não têm significado fora da estrutura da relação que os contém, seria mais preciso olhar para a construção primitiva não como uma dependência funcional, mas como uma relação binária. O processo de síntese é portanto o de construir relações *n*-árias a partir de relações binárias, com a restrição de que todas as relações montadas estejam na terceira forma normal. (As formas normais mais elevadas não estavam definidas quando este trabalho foi preparado.) Uma objeção mais séria (reconhecida pelo autor) é a de que as manipulações efetuadas pelo algoritmo de síntese são puramente sintáticas em sua natureza, não levando em conta a semântica. Por exemplo, dada as dependências funcionais

$$\begin{array}{l} R.A \rightarrow R.B \\ S.B \rightarrow S.C \\ T.A \rightarrow T.C \end{array}$$

a terceira pode ser ou não redundante (dedutível da primeira e segunda), dependendo do significado de R, S, T. Como um exemplo de caso em que não é, seja A um número de empregado, B um número de escritório, C um número de departamento; seja R o “escritório do empregado”, S o “departamento que possui o escritório”, e T o “departamento do empregado”; e consideremos o caso de um empregado trabalhando em um escritório pertencente a um departamento que não seja o seu. O algoritmo de síntese efetivamente supõe que S.C e T.C são um e o mesmo; ele confia na existência de um mecanismo externo (isto é, intervenção humana) para evitar manipulações semanticamente inválidas. No caso em questão, seria responsabilidade da pessoa que defina as FDs originais usar nomes distintos de atributos (digamos C e D) no lugar de S.C e T.C (de fato, nomes de relações tais como S e T não são reconhecidas pelo algoritmo).

- 14.17 J. Biskup, U. Dayal, and P. A. Bernstein. “Synthesizing Independent Database Schemas.” *Proc. 1979 ACM SIGMOD International Conference on Management of Data*.

Uma extensão do trabalho de [14.16].

- 14.18 R. Fagin. “The Decomposition Versus the Synthetic Approach to Relational Database Design.” *Proc. 3rd International Conference on Very Large Data Bases* (1977).

- 14.19 R. Fadous and J. Forsyth. “Finding Candidate Keys for Relational Data Bases.” *Proc. 1975 ACM SIGMOD International Conference on the Management of Data*.

Apresenta um algoritmo para encontrar todas as chaves candidatas em uma relação INF, dado o conjunto de todas as dependências funcionais naquela relação.

- 14.20 J. Rissanen. “Theory of Relations for Databases – A Tutorial Survey.” *Proc. 7th Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 64. New York: Springer-Verlag (1978).

- 14.21 C. Beeri, P. A. Bernstein, and N. Goodman. “A Sophisticate’s Introduction to Database Normalization Theory.” *Proc. 4th International Conference on Very Large Data Bases* (1978).

- 14.22 J. M. Cadiou. “On Semantic Issues in the Relational Model of Data.” *Proc. International Symposium on Mathematical Foundations of Computer Science*, Gdansk, Poland (setembro de 1975). New York: Springer-Verlag, Lecture Notes in Computer Science.

- 14.23 H. A. Schmid and J. R. Swenson. “On the Semantics of the Relational Data Model”. *Proc. 1975 ACM SIGMOD International Conference on Management of Data*.

Partindo da premissa de que o mundo real pode ser modelado como um “complexo de objetos independentes” (entidades) e associações entre eles, este artigo apresenta uma teoria de inserção/remoção para bancos de dados relacionais. Em termos do exemplo de fornecedores e peças, a teoria formaliza restrições tais como: (a) uma tupla SP só pode ser criada se o fornecedor indicado e a peça já existirem, e (b) uma tupla S ou P só pode ser removida se o fornecedor indicado ou a peça não participarem de qualquer associação SP. O autor enfatiza a distinção entre “associações” (como SP) e “características”. Uma característica pode ser representada por um atributo único – STATUS, por exemplo, é uma característica única de fornecedor – ou por uma relação subordinada – por exemplo, em um banco de dados de pessoal, podemos ter as relações:

```
EMP (EMP#, DESCRIÇÃO
      EC (EMP#, CARRO#
      CARRO(CAR#, DESCRIÇÃO
```

Aqui empregados são os “objetos complexos independentes” (estamos supondo para fins deste exemplo que o sistema não tem interesse em outros carros além dos de propriedade dos empregados); cada empregado está representado por uma tupla EMP, *n* tuplas EC e *n* tuplas CARRO. Carros são “características complexas” de empregados. Usando essas idéias, os autores categorizam as relações 3NF em cinco tipos diferentes, e recomendam que esta categorização seja refletida no esquema conceitual.

14.24 W. Kent. "Consequences of Assuming a Universal Relation." Será publicado em *ACM Transactions on Database Systems*. A abordagem de decomposição (veja por exemplo [14.18]) supõe que seja possível definir uma "relação universal" inicial (envolvendo todos os atributos relevantes do banco de dados considerado), e então mostra que aquela relação pode ser substituída por projeções cada vez menores até que seja alcançada uma "boa" estrutura. Este artigo sugere que esta suposição inicial não é realista, sendo de difícil justificativa tanto no plano prático como no teórico.

14.25 J. M. Smith. "A Normal Form for Abstract Syntax." *Proc. 4th International Conference on Very Large Data Bases* (1978).

14.26 R. Fagin. "A Normal Form for Relational Databases That Is Based on Domains and Keys." IBM Research Report RJ2520 (revised version, novembro de 1980). A ser publicado na *ACM Transactions on Database Systems*.

14.27 Y. Sagiv, C. Delobel, D. S. Parker, and R. Fagin. "An Equivalence Between Relational Database Dependencies And a Subclass of Propositional Logic." *JACM* (a ser publicado). Combina [14.12] e [14.15].

Parte 3

A Abordagem Hierárquica

Muitos sistemas atuais de bancos de dados estão baseados na abordagem hierárquica. Na Parte 3 examinaremos um desses sistemas, o IMS, com grande quantidade de detalhes. O Capítulo 15 descreve a estrutura global de um sistema IMS. Os Capítulos 16, 17 e 18 descrevem a estrutura de dados IMS (nos níveis “conceitual” e externo) e a linguagem DL/I de manipulação de dados. O Capítulo 19 está voltado para as partes de um sistema IMS que se encontram abaixo do interface com o usuário, isto é, a estrutura de armazenamento e o mapeamento entre os níveis conceitual e interno. Os Capítulos 20 e 21 fornecem introduções a dois dispositivos especiais do IMS, que são “bancos de dados lógicos” (Capítulo 20) e indexação secundária (Capítulo 21). Finalmente, o Capítulo 22 discute aspectos do banco de dados relacionados com o dispositivo IMS Fast Path. Embora não fazendo realmente parte da abordagem hierárquica, sendo definitivamente dispositivos especiais do IMS, incluímos esses últimos tópicos como exemplos interessantes de como a abordagem hierárquica pode ser estendida, e também de suas limitações.

Vários outros sistemas hierárquicos estão descritos nas referências do Capítulo 1.

15

A Arquitetura de Um Sistema IMS

15.1 HISTÓRICO

O nome IMS é a sigla do “Information Management System”. O IMS é um programa produto da IBM projetado para suportar programas de aplicação *batch* e *on-line*. Diversas versões distintas foram colocadas disponíveis pela IBM durante a evolução do IMS e produtos correlatos; as principais são o IMS/360 Versão 1, IMS/360 Versão 2, e a versão corrente à época do preparo deste livro, o IMS/VIS (Information Management System/Virtual Storage) Versão 1, que roda sob o sistema operacional IBM OS/VS (Operating System/Virtual Storage). Normalmente estaremos discutindo somente a versão mais recente; os manuais invariavelmente se referem a esta versão como IMS/VIS, mas nós usaremos a forma abreviada “IMS” no decorrer deste livro. Escolhemos o IMS como principal exemplo da abordagem hierárquica, pois até este momento ele é um dos mais extensamente usados de todos os sistemas de bancos de dados, hierárquicos ou de outros tipos.

Na sua forma básica, o IMS fornece apenas meios para se correr aplicações *batch*. É possível a extensão desse sistema, por meio do dispositivo de comunicação de dados, para permitir o desenvolvimento de aplicações *on-line* (isto é, aplicações que suportam acesso ao banco de dados a partir de terminais remotos); entretanto, essas aplicações usarão os mesmos meios das aplicações *batch* para acesso real ao banco de dados — o dispositivo de comunicação de dados fornece o suporte para acesso ao terminal, não ao banco de dados. Portanto, na maior parte, vamos ignorar os aspectos *on-line* do IMS, já que eles não tomam parte no sistema de *banco de dados* IMS.

15.2 ARQUITETURA

A arquitetura de um sistema IMS está ilustrada na Fig. 15.1. Observe antes de mais nada que os dados armazenados consistem de diversos bancos de dados, não apenas um. O banco de dados IMS é uma representação armazenada de um “banco de dados físico”, e um banco de dados físico, usando a terminologia relacional, é (mais ou menos) simplesmente uma relação não-normalizada — isto é, consiste de uma coleção de registros hierárquicos.

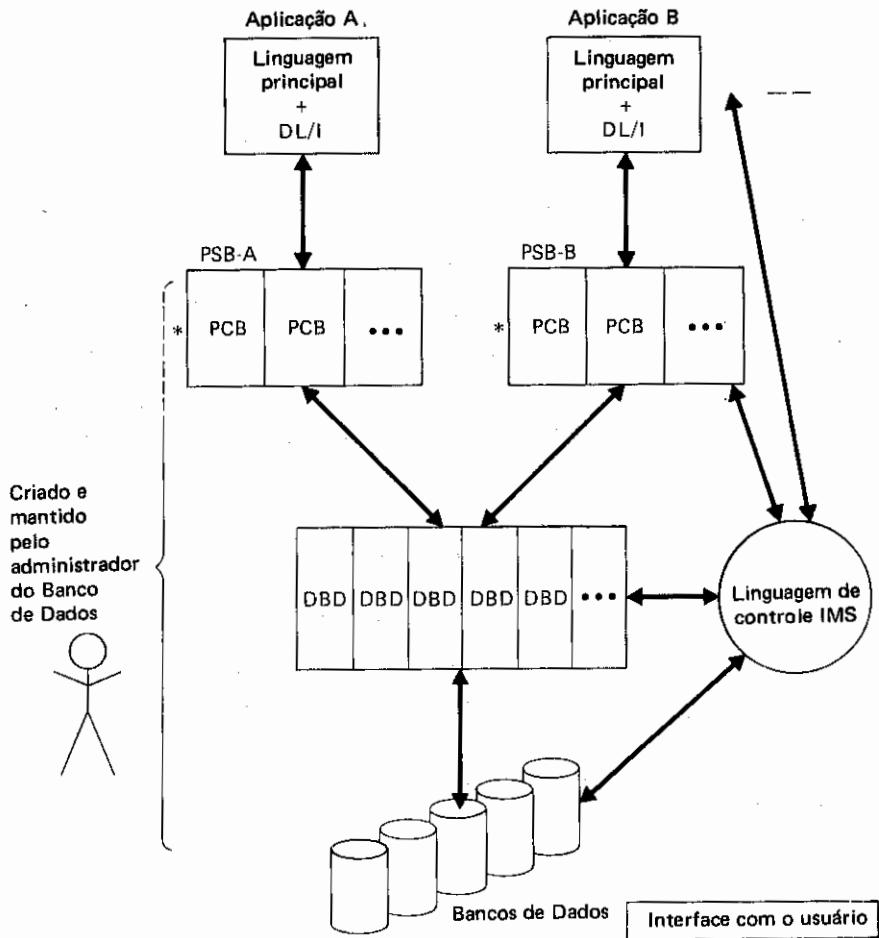


Fig. 15.1 Arquitetura de um sistema IMS.

A “visão conceitual” (não é um termo IMS) consiste então de uma coleção de bancos de dados físicos. O termo “físico” não é muito exato neste contexto, pois o usuário não vê esse banco de dados exatamente como ele está armazenado; na verdade, o IMS provê um grau bastante elevado de isolamento entre o usuário e a estrutura de armazenamento (e consequentemente um alto grau de independência de dados), como veremos no Capítulo 19. Cada banco de dados físico é definido por um *database description* (DBD – descrição do banco de dados). O mapeamento entre o banco de dados físico e o armazenamento é também especificado (pelo menos em parte; veja o Capítulo 19) no DBD. Assim, o conjunto de todos os DBDs corresponde ao esquema conceitual mais (parte da) definição associada ao mapeamento conceitual/interno.

Como na arquitetura genérica do Capítulo 1, o usuário não opera diretamente com o nível de banco de dados físico, mas sim sobre uma “visão externa” (não é um termo

IMS) dos dados. A visão externa específica de um usuário consiste de uma coleção de "bancos de dados lógicos", na qual cada banco de dados lógico é um subconjunto (em um sentido a ser explicado) do banco de dados físico correspondente.¹ Cada banco de dados lógico é definido, juntamente com seu mapeamento ao banco de dados físico, por meio de um *programming communication block* (PCB – bloco de comunicação com o programa). O conjunto de todos os PCBs de um usuário, correspondendo ao esquema externo mais a definição de mapeamento associada, é chamado de um *program specification block* (PSB – bloco de especificação do programa).

Finalmente, como mencionamos na Seção 15.1, os usuários são normalmente programadores de aplicações que usam uma linguagem principal (PL/I, COBOL, ou Assembler/370) através das quais pode ser invocada a linguagem DL/I de manipulação de dados do IMS – "Data Language I" – por meio de sub-rotinas de chamada. (Linguagem de manipulação de dados" não é um termo IMS.) Os *usuários finais* são suportados (num sistema que possua o dispositivo de comunicação de dados) por meio de programas de aplicação *on-line* escritos pelo usuário. O IMS não fornece uma linguagem de consulta integrada.

REFERÉNCIAS E BIBLIOGRAFIA

- 15.1 IBM Corporation. Information Management System/Virtual Storage General Information Manual. IBM Form nº GH20-1260.
- 15.2 D. C. Tsichritzis and F. H. Lochovsky. "Hierarchical Data Base Management: A Survey." *ACM Comp. Surv.* 8, nº 1 (março de 1976).
Inclui um breve estudo não só sobre IMS mas também sobre o System 2000.
- 15.3 W. C. McGee. "The IMS/VS System." *IBM Sys. J.* 16, nº 2 (junho de 1977).
Um extenso estudo sobre aspectos de banco de dados e comunicação de dados.
- 15.4 D. Kapp and J. F. Leben. *IMS programming Techniques: A Guide to Using DL/I*. New York: Van Nostrand Reinhold (1978).

¹ O termo "banco de dados lógico" tem realmente dois significados distintos no IMS. O aqui apresentado é provavelmente o menos importante dos dois. O segundo significado será introduzido no Capítulo 20. Devemos enfatizar que bancos de dados lógicos, conforme o segundo significado, não estão considerados antes do Capítulo 20; isto traz como efeito uma simplificação na apresentação da maior parte do material dos Capítulos 15–19. Diversos tópicos introduzidos nesses capítulos recebem uma interpretação estendida no Capítulo 20.

A *indexação secundária* também tem um efeito significativo sobre o nível externo. Por motivos didáticos, novamente, estaremos ignorando substancialmente todos os aspectos desse tópico até que alcancemos o Capítulo 21.

16

A Estrutura de Dados IMS

16.1 BANCOS DE DADOS FÍSICOS

No Capítulo anterior, a visão conceitual do IMS foi definida como uma coleção de *bancos de dados físicos* (PDBs). Um banco de dados físico é um conjunto ordenado, cujos elementos consistem de todas as ocorrências de um tipo de *registro de banco de dados físico* (PDBR). Uma ocorrência de PDBR por sua vez consiste de um arranjo hierárquico de ocorrências de *segmentos de comprimento fixo*,¹ e uma ocorrência de segmento consiste de um conjunto de ocorrências de *campos* associados de comprimento fixo. A menor unidade de dado à qual se pode ter acesso em uma única operação DL/I é a ocorrência de campo (embora a maior parte das operações DL/I lide com segmentos contendo diversos campos, e não apenas um).

Como um exemplo, consideremos um PDB contendo informações sobre um sistema de educação interna de uma grande companhia industrial. A estrutura hierárquica deste PDB – isto é, o *tipo* de PDBR – está ilustrada na Fig. 16.1.

Neste exemplo estamos supondo que a companhia mantém um departamento de educação cuja função é ministrar vários cursos de treinamento. Cada curso é oferecido a diversos locais diferentes dentro da companhia. O PDB contém tanto os detalhes das ofertas existentes quanto das previstas para ocorrerem no futuro. Os detalhes são os seguintes:

- Para cada curso; número do curso (único), título do curso, descrição do curso, detalhes de cursos pré-requisito (se houver), e detalhes de todas as ofertas (passadas e planejadas);
- Para cada curso pré-requisito de um dado curso: número e título do curso;
- Para cada oferta de um dado curso: data, local, formato (tempo parcial ou integral), detalhes sobre todos os professores e detalhes sobre todos os estudantes;

¹ São também permitidos segmentos de comprimento variável, mas os detalhes estão fora do âmbito deste texto.

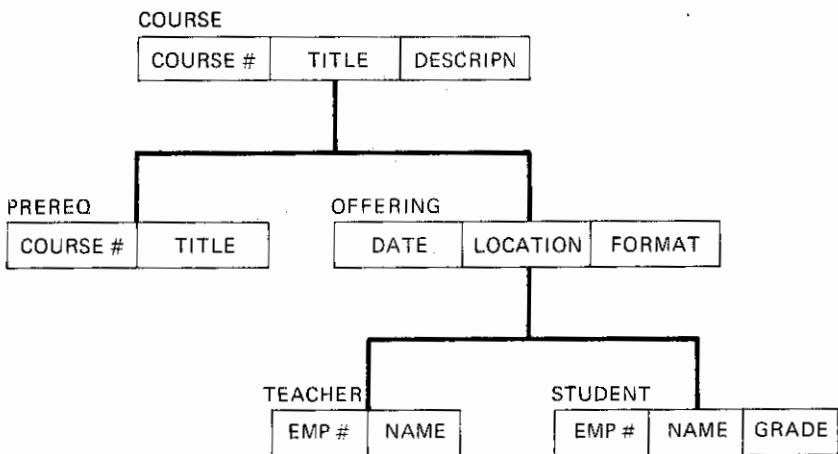


Fig. 16.1 Tipo de PDBR do banco de dados de educação.

- Para cada professor de uma dada oferta: número de empregado e nome;
- Para cada estudante de uma dada oferta: número de empregado, nome e grau.

Como mostra a Fig. 16.1, temos cinco tipos de segmentos: COURSE (curso), PREREQ (pré-requisito), OFFERING (oferta), TEACHER (professor) e STUDENT (estudante), cada um consistindo dos tipos de campos indicados. COURSE é o tipo de segmento *raiz*; os outros são tipos de segmentos dependentes. Cada dependente (tipo de segmento) tem um *pai* (tipo de segmento) – por exemplo, o pai de TEACHER (e STUDENT) é OFFERING. De forma semelhante, cada pai (tipo de segmento) tem pelo menos um *filho* (tipo de segmento); COURSE, por exemplo, tem dois.

É importante que se entenda que para uma ocorrência de dado tipo de segmento pode haver qualquer quantidade de ocorrências (possivelmente zero) de cada um dos tipos de ocorrências de seus segmentos filhos. A Fig. 16.2 ilustra este item.

Aqui nós temos uma ocorrência da raiz (COURSE) e portanto, por definição, uma ocorrência do tipo PDBR de educação. O PDB completo conterá diversas ocorrências PDBR, representando informações sobre diversos cursos. Na ocorrência PDBR especifica mostrada na Fig. 16.2, nós temos como dependentes da ocorrência COURSE duas ocorrências de PREREQ e três de OFFERING. A primeira ocorrência de OFFERING por sua vez possui uma ocorrência de TEACHER e diversas ocorrências de STUDENT sob sua dependência (só estão mostradas três). As outras ocorrências OFFERING não têm ainda professores ou estudantes designados a elas.

A nomenclatura *pai-filho*, introduzida anteriormente para os tipos de segmentos, também se aplica a ocorrências de segmentos. Assim, cada ocorrência de segmento dependente possui um *pai* (ocorrência de segmento) – o *pai* da ocorrência TEACHER (e das ocorrências STUDENT) é, por exemplo, a primeira ocorrência OFFERING. Inversamente, as ocorrências TEACHER e STUDENT são consideradas como *filhas* (ocorrências de segmentos filhos) desta ocorrência OFFERING. Em adição, todas as ocorrências de um determinado tipo de segmento que compartilhem uma ocorrência comum de *pai* são cha-

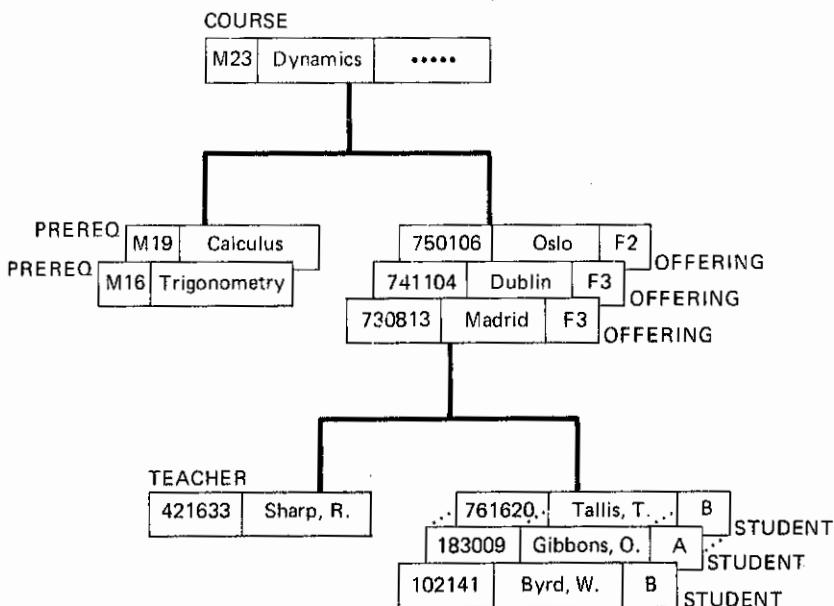


Fig. 16.2 Exemplo de ocorrência PDBR do banco de dados de educação.

madas de *gêmeas*. Por exemplo, as ocorrências de OFFERING na Fig. 16.2 são gêmeas (embora haja três delas).

Resumindo:

- Um tipo de PDBR contém um só tipo de segmento raiz.
- A raiz pode ter qualquer quantidade de tipos de segmentos filhos.
- Cada filho da raiz pode também ter qualquer quantidade de tipos de segmentos filhos, e assim por diante (até o máximo de 15 tipos de segmentos em qualquer caminho hierárquico, e um máximo de 255 tipos de segmentos no total do tipo PDBR).²
- Em uma ocorrência de qualquer tipo de determinado segmento pode haver qualquer quantidade de ocorrências (possivelmente zero) de cada um dos seus filhos.
- Não pode haver ocorrência de segmento filho sem seu pai. O último item é um reforço da filosofia hierárquica. Significa que, por exemplo, se for removida determinada ocorrência, também o serão todos os seus filhos (como explicamos no Capítulo 3).

A partir deste ponto não mais usaremos os termos “tipo” e “ocorrência”, desde que não haja possibilidade de confusão; na verdade já começamos a fazer isso, como o leitor deve ter percebido.

² Naturalmente esses valores são característicos do IMS, não um componente fundamental da abordagem hierárquica.

16.2 A DESCRIÇÃO DO BANCO DE DADOS

Cada banco de dados físico é definido, juntamente com seu mapeamento de memória, por meio de um *database description* (DBD – descrição do banco de dados). O formato fonte do DBD é escrito usando-se macroinstruções especiais da linguagem Assembler/370. (Essas macros constituem portanto a “DDL conceitual” do IMS.) Uma vez escrito, o DBD é montado, sendo o formato objeto guardado em uma biblioteca do sistema, de onde pode ser extraído quando solicitado pelo programa de controle do IMS.

Vamos por enquanto ignorar a parte do DBD relativa ao mapeamento de memória. Vamos também ignorar algumas instruções puramente dedicadas aos detalhes formais (tais como geração de mensagens de erro). O restante – que poderíamos chamar de “parte do esquema conceitual” – está mostrado no exemplo da Fig. 16.3 para o banco de dados de educação. As instruções estão numeradas para possibilitar a explicação a seguir.

Explicação

A instrução 1 somente designa o nome EDUCPCB (“descrição do banco de dados físico de educação”) à DBD. Em IMS, todos os nomes são limitados ao comprimento máximo de oito caracteres.

A instrução 2 define o tipo de segmento raiz como tendo o nome COURSE, com comprimento de 256 bytes.

As instruções 3–5 definem os tipos de campos que compõem COURSE. Cada um recebe um nome, um comprimento em bytes, e uma posição inicial dentro do segmento. O primeiro campo, COURSE#, é definido (via especificação SEQ) como sendo o campo de ordenação do segmento. Isto significa que dentro do PDB de educação as ocorrências PDBR estarão em ordem ascendente por número de curso.

A instrução 6 define PREREQ como sendo um segmento dependente de COURSE, com 36 bytes.

```
1 DBD      NAME=EDUCPDBD
2 SEGM     NAME=COURSE, BYTES=256
3 FIELD    NAME=(COURSE#, SEQ), BYTES=3, START=1
4 FIELD    NAME=TITLE, BYTES=33, START=4
5 FIELD    NAME=DESCRIPN, BYTES=220, START=37
6 SEGM     NAME=PREREQ, PARENT=COURSE, BYTES=36
7 FIELD    NAME=(COURSE#, SEQ), BYTES=3, START=1
8 FIELD    NAME=TITLE, BYTES=33, START=4
9 SEGM     NAME=OFFERING, PARENT=COURSE, BYTES=20
10 FIELD   NAME=(DATE, SEQ, M), BYTES=6, START=1
11 FIELD   NAME=LOCATION, BYTES=12, START=7
12 FIELD   NAME=FORMAT, BYTES=2, START=19
13 SEGM     NAME=TEACHER, PARENT=OFFERING, BYTES=24
14 FIELD   NAME=(EMP#, SEQ), BYTES=6, START=1
15 FIELD   NAME=NAME, BYTES=18, START=7
16 SEGM     NAME=STUDENT, PARENT=OFFERING, BYTES=25
17 FIELD   NAME=(EMP#, SEQ), BYTES=6, START=1
18 FIELD   NAME=NAME, BYTES=18, START=7
19 FIELD   NAME=GRADE, BYTES=1, START=25
```

Fig. 16.3 DBD (parte do esquema conceitual) do PDB de educação.

As instruções 7–8 definem os campos de PREREQ. O primeiro campo, COURSE# (novamente), está definido como o campo de ordenação de PREREQ. Isto significa que para cada ocorrência do pai (COURSE), as ocorrências do filho (PREREQ) estarão em ordem ascendente de número de curso. (Em outras palavras, o campo de ordenação de um tipo de segmento filho define “seqüência dos gêmeos” para aquele tipo de filho. O mesmo pode ser dito para o tipo de segmento pai, se considerarmos todas as ocorrências de raiz como gêmeas uma da outra.)

A instrução 9 define OFFERING como filha de COURSE.

As instruções 10–12 definem os campos de OFFERING. DATE está definido como o campo de ordenação de OFFERING. A especificação M (múltiplo) significa que as ocorrências gêmeas de OFFERING podem conter o *mesmo* valor de dado (significando neste caso que duas ofertas do mesmo curso estão sendo ministradas ao mesmo tempo).

As instruções 13–15 definem o segmento TEACHER (filho de OFFERING) e seus campos.

As instruções 16–19 definem o segmento STUDENT (filho de OFFERING) e seus campos.

A seqüência das instruções no DBD é significativa. Especificamente, as instruções SEGM têm que aparecer na seqüência que reflete a estrutura hierárquica (de cima para baixo, da esquerda para a direita);³ também cada instrução SEGM tem que estar imediatamente seguida pelas instruções FIELD apropriadas. Como iremos ver, o primeiro destes itens tem um efeito bastante sensível para o usuário. Significa que a ordenação, não somente de ocorrências de segmentos mas também de *tipos* de segmentos, está contida na própria estrutura dos dados, de tal forma que, por exemplo, o usuário pode emitir uma operação DL/I “obtenha o próximo” para ir de uma ocorrência de TEACHER para uma de STUDENT.

Alguns itens adicionais:

- A especificação de um campo de ordenação é opcional, exceto quanto ao que está mencionado abaixo.
- O campo de ordenação, se especificado, é considerado como *único*, a menos que seja especificado M (múltiplo). Aqui, o “único” significa que duas ocorrências de um determinado tipo de segmento sob uma ocorrência comum de pai – ou, no caso de raiz, duas ocorrências do tipo de segmento no banco de dados – não poderão ter o mesmo valor para o campo de ordenação.
- É exigido um campo de ordenação único no segmento raiz em HISAM e HIDAM (veja o capítulo 19).
- A regra dada anteriormente para seqüência gêmea (valores ascendentes do campo de ordenação) não especifica o que ocorre se o campo de ordenação for omitido ou não-único. Nesse caso, é necessária especificação adicional no DBD (não mostrada na Fig. 16.3), podendo ser necessária programação adicional por parte do usuário quando for inserido novo segmento. Além disso, em certas situações, a falta de um campo de ordenação único pode levar a sérias dificuldades de lógica [19.1]; os deta-

³

Internamente, o IMS identifica cada tipo de segmento por sua posição na estrutura hierárquica. Assim, no PDB de educação, COURSE tem o código de tipo 1, PREREQ tem o código de tipo 2, OFFERING 3, TEACHER 4, e STUDENT 5.

Ihes estão além do escopo deste livro, mas por essas razões geralmente nos restringiremos a campos únicos de ordenação no decorrer deste livro.

- A instrução FIELD do campo de ordenação, se houver, tem que ser a primeira instrução do segmento.
- Pode ser definida superposição de campos; por exemplo, o segmento COURSE pode ser definido contendo um campo COURSE# N (BYTES = 2, START = 2), representando o segundo e terceiro caracteres numéricos do campo COURSE#. [Observe que isto possibilita que seja definida a combinação de diversos campos (contíguos) como campo de ordenação.]
- A instrução FIELD pode, opcionalmente, incluir a especificação “TYPE = tipo de dado”, onde “tipo de dado” pode ser C (caracter), X (hexadecimal), ou P (compactado). C é o tipo assumido se não houver especificação. No entanto, o IMS *não* verifica a validade dos valores dos campos (exceto no que está mencionado no Capítulo 22), e as operações DL/I sempre executam comparações *bit a bit* da esquerda para a direita (novamente, com exceção do que está mencionado no Capítulo 22).

16.3 SEQUÊNCIA HIERÁRQUICA

O conceito de seqüência hierárquica dentro de um banco de dados é muito importante no IMS. Podemos defini-la formalmente como se segue:⁴

- Para cada ocorrência de segmento, definimos o “valor da chave de seqüência hierárquica” consistindo do valor do campo de ordenação daquele segmento, prefixado pelo código do tipo de segmento (veja nota 3 de rodapé), prefixado pelo valor da chave de seqüência hierárquica de seu pai, se existir. Por exemplo, o valor da chave de seqüência hierárquica da ocorrência STUDENT para “Byrd, W.” (veja Fig. 16.2) é:

1M2337308135102141

Assim, a seqüência hierárquica de um banco de dados é a das ocorrências dos segmentos definida pelos valores ascendentes da chave de seqüência hierárquica.

A razão para a importância desta noção é que os bancos de dados IMS são armazenados, em geral, em seqüência hierárquica;⁵ consequentemente algumas operações DL/I – essencialmente as voltadas à recuperação seqüencial e de carga do banco de dados – são definidas em termos dessa seqüência. (Esta afirmativa não é muito verdadeira para HDAM; veja o Capítulo 19.)

16.4 ALGUMAS OBSERVAÇÕES SOBRE O BANCO DE DADOS DE EDUCAÇÃO

Antes de sairmos do assunto da estrutura de dados IMS, há alguns itens adicionais a serem vistos com relação ao exemplo de educação. Eles provêm das diversas redundâncias deliberadamente introduzidas no exemplo.

⁴ Não estamos considerando o caso em que os campos de ordenação são omitidos ou não-únicos.

⁵ Esta afirmativa não deve ser entendida como significando que os dados são *fisicamente* armazenados nessa seqüência. Não verdade são usadas diversas técnicas – encadeamento, indexação e outras – para representar a seqüência hierárquica. Veja detalhes no Capítulo 19.

Consideremos inicialmente os estudantes. Para cada oferta de curso da qual participa determinado estudante, o PDB conterá o número de empregado desse estudante e seu nome. Dessa forma, a associação entre um determinado número e o respectivo nome aparecerá em geral diversas vezes. Por sua vez, isto introduz a possibilidade de que um determinado número de empregado tenha nomes diferentes associados a ele em pontos diferentes, isto é, que o PDB possa estar inconsistente.

As mesmas observações aplicam-se a professores, onde novamente a redundância envolve números e nomes de empregados, e a cursos pré-requisito. Por fim, a redundância envolve o relacionamento entre números e títulos de cursos; não só aparece um determinado evento desta associação cada vez que certo curso é pré-requisito para algum outro, mas também na raiz da ocorrência PDBR do próprio curso.

Entretanto, pode ser desejável fornecer ao usuário uma estrutura de dados contendo redundâncias como essas. O dispositivo de “banco de dados lógico” do IMS permite que haja redundância nos dados percebidos pelo usuário *sem* necessariamente introduzir a correspondente redundância nos dados armazenados, e portanto *sem* possibilidades de inconsistência. (“Banco de dados lógico” está sendo usado aqui no sentido do Capítulo 20). Para ver por que a redundância pode ser desejável, observemos o que aconteceria se ela fosse removida deste exemplo.

Para remover a redundância de nome de empregado, podemos eliminar o campo de nome de empregado dos segmentos TEACHER e STUDENT, e introduzir um outro PDB contendo um só tipo de segmento — digamos EMP — com campos EMP# e NAME. Este PDB representa a necessária associação entre números e nomes de empregados. De maneira semelhante, podemos eliminar o campo de título de curso do segmento PREREQ; a associação entre números e títulos de cursos já está representada no PDB de educação (no segmento COURSE). Entretanto, o efeito de remover as redundâncias dessa forma é o de provocar mais trabalho para o usuário, como explicaremos no parágrafo seguinte.

Observemos primeiro que a estrutura de dados vista pelo usuário torna-se mais complicada, pois agora contém dois PDBs (interconectados) ao invés de um. Por natureza do DL/I, o usuário só pode processar um banco de dados por operação. Assim, a programação requerida em muitos casos irá torna-se mais complexa do que antes. Consideremos por exemplo a consulta “Encontre os nomes de todos os estudantes de uma oferta específica de determinado curso”. Sob a estrutura original, a tarefa era comparativamente direta, envolvendo somente uma varredura de todas as ocorrências STUDENT subordinadas à OFFERING especificada do COURSE especificado. Na estrutura revista, o usuário ainda tem que fazer a varredura do mesmo conjunto de ocorrências STUDENT; entretanto, além disso, terá que extrair o valor EMP# para cada STUDENT, usando-o para recuperar a ocorrência EMP do segundo banco de dados.

Como outro exemplo, vejamos a consulta “Encontre os títulos de todos os cursos pré-requisito para um determinado curso”. Sob a estrutura original, esta consulta requer apenas uma varredura de todas as ocorrências PREREQ subordinadas ao COURSE específico. Na estrutura revista, irá requerer essa varredura, seguida da extração dos valores de COURSE# e operações adicionais de recuperação (sobre o mesmo banco de dados) — com complicações adicionais resultantes da noção IMS de posição corrente (a ser explicada no Capítulo 18).

Por razões como essas, é usual escolher-se uma estrutura de dados IMS envolvendo alguma redundância. Como já mencionamos, isto não implica redundância nos dados armazenados.

EXERCÍCIOS

16.1 Escreva as ocorrências de segmentos da Fig. 16.2 em seqüência hierárquica.

16.2 A Fig. 16.4 representa a estrutura hierárquica de um banco de dados físico contendo informações sobre publicações a respeito de várias áreas de interesse.

Para as finalidades deste banco de dados, vamos supor que as publicações são de dois tipos: artigos e monografias. Um artigo, sendo notícia que apareceu em um jornal ou revista que, em geral, também contém outros artigos (de autores diferentes). Uma monografia, sendo uma publicação totalmente dedicada ao trabalho de um autor ou time de autores. Observe que uma publicação aparecerá em diversos lugares diferentes; por exemplo, o mesmo artigo pode ter sido publicado tanto pela *Comunicação da ACM* como pelo *Boletim de Computação BCS* (a referência [1.3] é um exemplo).

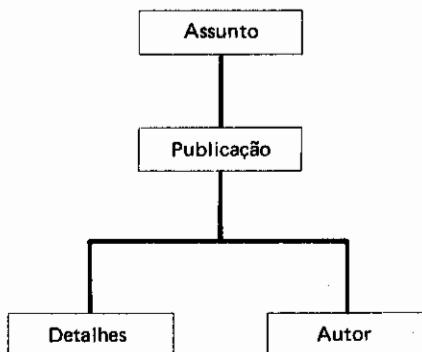


Fig. 16.4 Tipo de PDBR do banco de dados de publicações.

Os segmentos contêm os seguintes campos:

- **Assunto:** número de classificação do assunto (único), nome do assunto.
- **Publicação:** indicador de tipo (A = artigo, M = monografia), título.
- **Detalhes:** data da publicação, editor (se monografia), nome do jornal, número do volume e do exemplar (se artigo).
- **Autor:** nome e endereço do autor.

Escreva um DBD apropriado (a parte de esquema conceitual).

REFERÊNCIAS E BIBLIOGRAFIA

16.1 IBM Corporation. Information Management System/Virtual Storage Utilities Reference Manual. IBM Form n° SH20-9029.

Este manual traz detalhes completos sobre especificação do DBD (e também especificação do PSB; veja o Capítulo 17).

17

O Nível Externo do IMS

17.1 BANCOS DE DADOS LÓGICOS

No Capítulo 15 definimos a visão externa de um determinado usuário como sendo uma coleção de bancos de dados lógicos e, por sua vez, banco de dados lógico foi definido como um subconjunto de banco de dados físico (único) correspondente. Podemos agora ampliar essas colocações. Um banco de dados lógico (LDB) é um conjunto ordenado, cujos elementos consistem de todas as ocorrências de um tipo de registro de banco de dados lógico (LDBR). Um tipo LDBR é um arranjo hierárquico de tipos de segmentos; esta hierarquia deriva da hierarquia PDBR correspondente de acordo com as seguintes regras:

1. Qualquer tipo de segmento da hierarquia PDBR, juntamente com todos os seus dependentes (em todos os níveis), pode ser omitido da hierarquia LDBR.¹
2. Os campos de um tipo de segmento LDBR podem ser um subconjunto dos do tipo de segmento PDBR correspondentes, e podem ser rearranjados dentro do tipo daquele segmento LDBR.

A regra 1 implica em que a raiz LDB tenha que ser a mesma raiz PDB. Consideremos o PDB de educação mostrado na Fig. 16.1. Ignorando a Regra 2, existem basicamente dez LDBs distintos que podem ser derivados daquele PDB. Um deles está mostrado na Fig. 17.1. Quais são os outros?

Segmentos Sensitivos

Os tipos de segmentos PDB incluídos nos segmentos LDB – COURSE, OFFERING, e STUDENT na Fig. 17.1 – são ditos serem “sensitivos”. Um usuário deste LDB não estará

¹ É também possível reordenar tipos de segmentos filhos sob um tipo de segmento pai (isto é, ter OFFERINGS aparecendo à esquerda dos PREREQS sob COURSES), se esses filhos estiverem conectados na memória aos seus pais via “indicadores de localização filhos/gêmeos” (veja o Capítulo 19).

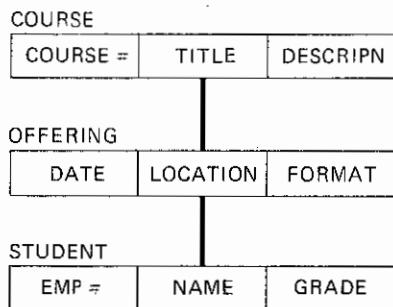


Fig. 17.1 Exemplo do tipo LDBR do banco de dados de educação.

a par da existência de quaisquer outros segmentos; por exemplo, a operação DL/I “obtenha o próximo”, geralmente usada para recuperação seqüencial, simplesmente saltará sobre quaisquer segmentos não-sensitivos para aquele usuário. A única exceção ocorre se o usuário remover um segmento sensitivo (ocorrência), pois então todos os filhos daquele segmento também serão removidos, independentemente de serem sensitivos ou não. Na prática, um usuário provavelmente não terá autorização para remover um segmento se isto causar a remoção também de segmentos escondidos (veja a discussão sobre PROCOPT na Seção 17.2).

O conceito de segmento sensitivo protege o usuário de alguns tipos de crescimento do PDB. Especificamente, pode ser adicionado um novo tipo de segmento (como filho de um segmento existente) em qualquer ponto, bastando que isto não afete de qualquer forma algum relacionamento pai-filho existente.² O novo segmento simplesmente não será sensitivo para nenhum usuário existente. No caso do PDB de educação, pode ser adicionado um segmento novo em qualquer dos seguintes pontos:

1. Subordinado a PREREQ
2. Subordinado a TEACHER
3. Subordinado a STUDENT
4. Subordinado a COURSE (no mesmo nível de PREREQ e OFFERING)
5. Subordinado a OFFERING (no mesmo nível de TEACHER e STUDENT)

Entretanto, a introdução de um novo segmento entre COURSE e OFFERING afetaria definitivamente os usuários atuais daquele caminho hierárquico específico.

O conceito de segmento sensitivo também fornece um grau de controle sobre segurança de dados, no sentido de que os usuários podem ser impedidos de ter acesso a tipos específicos de segmentos, pela omissão desses segmentos no seu LDB (exceto no que mencionamos acima sobre remoções).

² Naturalmente, terá que ser criado um novo PDB e, na maioria dos casos, o PDB terá que ser descarregado e recarregado de acordo com o novo DBD. (Não será necessária recarga se o novo segmento aparecer depois de todos os segmentos existentes na seqüência de cima para baixo, da esquerda para a direita.)

Campos Sensitivos

Campos sensitivos são aqueles campos (isto é – tipos de campos) do PDB que estão incluídos no LDB. Por definição, cada campo sensitivo tem que estar contido dentro de um segmento sensitivo. Um determinado LDB pode em geral excluir qualquer combinação de campos do PDB, exceto que, se o programa pretender inserir novas ocorrências de um dado tipo de segmento, terá então que ser “sensitivo” ao campo de ordenação daquele tipo de segmento.

A sensitividade de campo, como a sensitividade de segmento, protege o usuário de algum tipo de crescimento do banco de dados e fornece um nível simples de segurança de dados.

17.2 O PROGRAM COMMUNICATION BLOCK

Cada banco de dados lógico é definido por um bloco de comunicação com o programa (PCB). O PCB inclui a especificação do mapeamento entre o LDB e o PCB correspondente (que é naturalmente muito simples). Como o DBD, um PCB é escrito usando-se macroinstruções especiais da linguagem Assembler/370. Essas instruções constituem a “DDL externa” do IMS. O conjunto de todos os PCBs de um determinado usuário forma o bloco de especificação do programa (PSB) daquele usuário; o formato objeto é armazenado em uma biblioteca do sistema, da qual pode ser extraído quando solicitado pelo programa de controle IMS.

A Fig. 17.2 mostra o PCB do LDB da Fig. 17.1. Novamente as instruções estão numeradas para fins das explicações que se seguem.

```
1 PCB      TYPE=DB,DBDNAME=EDUCPDBD,KEYLEN=15
2 SENSEG   NAME=COURSE,PROCOPT=G
3 SENSEG   NAME=OFFERING,PARENT=COURSE,PROCOPT=G
4 SENSEG   NAME=STUDENT,PARENT=OFFERING,PROCOPT=G
```

Fig. 17.2 PCB do LDB da Fig. 17.1.

Explicação

A instrução 1 especifica (a) que este PCB é um PCB banco de dados, não um PCB terminal;³ (b) que o DBD do banco de dados físico básico chama-se EDUCPCB; e (c) que o comprimento da área de recebimento da chave é de 15 bytes. O último item requer alguma explicação. Quando o usuário faz acesso ao LDB, o PDB correspondente é mantido na memória e atua como uma área de comunicação entre o programa do usuário e o IMS. Um dos campos do PCB é a área para recebimento da chave. Quando o usuário recupera um segmento do LDB (via uma operação “get” do DL/I), o IMS não somente obtém o segmento pedido como também coloca uma “chave totalmente concatenada” na área de recebimento da chave. A chave totalmente concatenada consiste da concatenação dos valores dos campos de ordenação de todos os segmentos do caminho hierárquico desde a

³ PCBs terminais são usadas em conexão com o dispositivo de comunicação de dados.

raiz até o segmento recuperado. Por exemplo, se o usuário recuperar a ocorrência STUDENT de "Byrd, W." (veja Fig. 16.2), o IMS colocará o valor:

M23730813102141

na área de recebimento da chave. Para que o IMS possa reservar uma área de recebimento da chave suficientemente grande dentro do PCB, o comprimento máximo para uma chave totalmente concatenada (considerando todos os caminhos hierárquicos no LDB) pode ser calculado e lançado na entrada KEYLEN da instrução PCB. No nosso exemplo o valor é 15 (3 para COURSE#, mais 6 para DATE, mais 6 para EMP#).

Observe que a chave totalmente concatenada de um segmento não é exatamente a mesma coisa que a "chave de seqüência hierárquica" da Seção 16.3 – ela não inclui a informação do código de tipo de segmento.

A instrução 2 especifica o primeiro segmento sensitivo (a raiz) no LDB. Observe que o nome do segmento sensitivo tem que ser o mesmo dado ao segmento no DBD. A entrada PROCOPT ("processing options" – opções de processamento) especifica os tipos de operações que o usuário está autorizado a executar neste segmento. No exemplo, a entrada é G ("get" – obtenha), indicando somente recuperação. Outros valores possíveis são I ("insert" – inserção), R ("replace" – substituição), e D ("delete" – remoção); podem ser especificados qualquer um ou todos os G, I, R, e D, em qualquer ordem. A referência [16.1] explica algumas outras entradas.

A instrução 3 define o próximo segmento sensitivo do LDB. Como no DBD, os segmentos são especificados em seqüência hierárquica (de cima para baixo, da esquerda para a direita). A entrada PARENT especifica o segmento pai apropriado (tem que ser como definido no DBD). Novamente foi lançado G no PROCOPT.

A instrução 4 define o último segmento sensitivo. No nosso exemplo, as instruções 3 e 4 são muito parecidas. No PCB da Fig. 17.1 a entrada PROCOPT é a mesma para cada um dos três segmentos sensitivos. Nessa situação, podemos especificar PROCOPT na instrução PCB ao invés de em cada instrução SENSEG. Se for especificado PROCOPT na instrução PCB e na instrução SENSEG, a entrada na instrução SENSEG geralmente modifica a entrada da instrução PCB. No entanto, há uma entrada, L ("load" – carga – isto é, crie a versão inicial do banco de dados), que só pode ser especificada na instrução PCB e não pode ser modificada. Há também uma entrada, K ("key sensitivity" – sensitividade de chave) que só pode ser especificada na instrução SENSEG. K é usada quando o projetista do PCB é forçado pela estrutura hierárquica do banco físico básico de dados a incluir um segmento que o usuário do PCB realmente não requer (ou talvez não esteja autorizado a fazer acesso). Suponhamos, por exemplo, que uma determinada aplicação interessa-se somente por cursos e estudantes, mas não por ofertas. O PCB deste usuário tem que incluir o segmento OFFERING, pois este faz parte do caminho hierárquico entre COURSE e STUDENT. Entretanto, se PROCOPT = K for especificado na instrução de OFFERING, o usuário poderá praticamente ignorar a presença de OFFERING na hierarquia. Em outras palavras, na maioria dos casos o usuário poderá imaginar o banco de dados lógico como se este tivesse a estrutura mostrada na Fig. 17.3; a principal diferença é que quando for recuperada uma ocorrência STUDENT, a chave totalmente concatenada na área de recebimento da chave incluirá o valor de dado do pai OFFERING. Mais precisamente, o usuário pode emitir solicitações DL/I para recuperação exatamente como se os segmentos com sensitividade de chave *não* estivessem ausentes – no caso presente, especificamente, o usuário pode fazer referência a OFFERING e a seus campos da forma

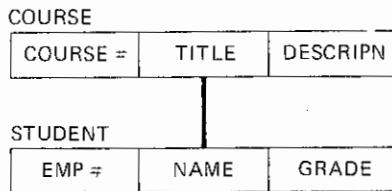


Fig. 17.3 Efeito da especificação PROOPT = K para OFFERING.

usual – e o IMS irá manusear essas solicitações exatamente da forma normal,⁴ até o momento em que os dados recuperados (se houver) estejam prestes a ser entregues ao programa. Nesse ponto, a entrega será suprimida para qualquer segmento cuja sensitividade de chave tenha sido especificada (embora o valor do campo de ordenação do segmento seja colocado na área de recebimento da chave). Um segmento com sensitividade de chave não pode servir como objetivo para uma operação DL/I de atualização (inserção, remoção, ou substituição), muito embora seja naturalmente possível remover tal segmento removendo o seu pai.

O LDB da Fig. 17.1 (como definido pelo PCB da Fig. 17.2) é sensitivo em todos os campos dos segmentos COURSE, OFFERING, e STUDENT do PDB básico. Suponhamos que queremos excluir o campo LOCATION do segmento OFFERING do LDB, mantendo a sensitividade de todos os outros campos da Fig. 17.1. O PCB da Fig. 17.2 tem que ser estendido para incluir as instruções:

```
SENFLD NAME=FORMAT, START=1
SENFLD NAME=DATE, START=3
```

imediatamente após a instrução SENSEG de OFFERING. Essas instruções especificam os campos a serem incluídos no segmento do LDB e sua posição inicial naquele segmento. (Sómente para efeito de exemplo, intercambiamos os dois campos sensitivos dentro do segmento.) Os nomes e comprimentos desses campos são herdados do DBD. Se não for fornecida instrução SENFLD para a instrução SENSEG determinada, o segmento será considerado automaticamente como idêntico ao segmento do PCB básico (a menos que PROOPT = K).

EXERCÍCIO

17.1 Escreva um PCB (com todos os segmentos e campos sensitivos) para o PDB de publicações (veja Exercício 16.2). Este PCB só deve ser usado para operações de recuperação.

REFERÊNCIAS E BIBLIOGRAFIA

Veja [16.1].

⁴ Uma exceção a esta afirmativa é que solicitações de recuperação sem SSAs – veja os Exemplos 18.3.6 e 18.3.9 no próximo capítulo – saltarão sobre segmentos com chaves sensitivas.

18

Manipulação de Dados IMS

18.1 DEFININDO O PROGRAM COMMUNICATION BLOCK (PCB)

A linguagem de manipulação de dados do IMS (DL/I) é invocada pelas linguagens principais (PL/I, COBOL, ou Linguagem Assembler do Sistema/370) por meio de sub-rotinas comuns de chamada. Como mencionamos na Seção 17.2, quando um programa de aplicação está operando sobre um determinado banco de dados lógico (LDB), o PCB daquele LDB é mantido na memória para servir como uma área de comunicação entre o programa e o IMS; de fato, quando o programa chama a DL/I, ele tem que indicar o endereço de memória do PCB apropriado para identificar à DL/I sobre que LDB ela irá operar (lembre-se de que um programa pode, em geral, ter acesso a diversos LDBs).

Como então o programa conhece o endereço do PCB? A resposta é que este é fornecido ao programa pelo IMS quando o programa é carregado. O que ocorre é o seguinte: quando se torna necessário correr uma aplicação do banco de dados, primeiramente o IMS recebe o controle. O IMS determina que PSB e DBD(s) são exigidos, localiza-os em suas respectivas bibliotecas, e carrega-os na memória. Depois o IMS carrega o programa de aplicação a ele passa o controle, entregando os endereços dos PCBs como parâmetros.

Para que o programa de aplicação possa ter acesso à informação no PCB de um determinado LDB, ele tem que ter uma definição daquele PCB. A definição não irá reservar espaço de memória, mas sim atuar como uma máscara que se ajusta sobre o PCB real (que fisicamente não se encontra dentro da área de memória da aplicação). Uma referência feita a um campo da definição do PCB será interpretada como uma referência ao campo correspondente do PCB real.

Por exemplo, suponhamos que temos uma aplicação PL/I que opera sobre o LDB course-offering-student da Fig. 17.1, e também que este é o único LDB usado por esta aplicação. Então, parte do programa poderia se parecer com a Fig. 18.1.

Explicação

A instrução PROCEDURE (rotulada de DLITPLI) é o ponto de entrada do programa. O nome DLITPLI é mandatório (para PL/I); todos os outros nomes mostrados (COSPCB,

```
DLITPLI: PROCEDURE(COSPCB_ADDR) OPTIONS(MAIN);
```

```
DECLARE 1 COSPCB      BASED(COSPCB_ADDR),  
        2 DBDNAME    CHARACTER(8),  
        2 SEGLEVEL   CHARACTER(2),  
        2 STATUS     CHARACTER(2),  
        2 PROCOPT    CHARACTER(4),  
        2 RESERVED   FIXED BINARY(31),  
        2 SEGNAME    CHARACTER(8),  
        2 KEYFBLEN   FIXED BINARY(31),  
        2 #SENSEGS   FIXED BINARY(31),  
        2 KEYFBAREA  CHARACTER(15);
```

Fig. 18.1 Exemplo de início do programa e definição do PCB (PL/I).

DBDNAME, SEGLEVEL, etc.) são arbitrários. A expressão entre parêntesis que se segue à palavra chave PROCEDURE representa os parâmetros a serem passados ao programa pelo IMS; em geral, consistirá de uma lista de indicadores de localização, um para cada PCB no PSB, na qual o primeiro indicador de localização fornece o endereço do primeiro PCB, e assim por diante. No nosso exemplo, há somente um PCB e consequentemente apenas um indicador de localização na lista.

O restante da Fig. 18.1 consiste de instruções DECLARE que definem a estrutura (chamada COSPCB) que representa o único PCB usado por esta aplicação. Esta estrutura está baseada no indicador de localização COSPCB-ADDR. Os campos da estrutura são usados para receber diversas informações fornecidas pelo IMS, como se segue.

O campo DBDNAME contém o nome do DBD básico (EDUCPDBD no nosso exemplo) durante a execução do programa.

O campo SEGLEVEL é ajustado, após uma operação DL/I, para conter o número do nível do segmento que recebeu acesso (onde o segmento raiz tem o nível 1, seus filhos o nível 2, e assim por diante.)

O campo STATUS é sem dúvida o mais importante no PCB, do ponto de vista do usuário. Após cada chamada DL/I, é colocado um valor com dois caracteres neste campo, para indicar se a operação solicitada foi ou não bem-sucedida. Um valor em branco indica que a operação foi completada satisfatoriamente; qualquer outro valor representa uma condição excepcional ou de erro (por exemplo, um valor GE significa "segmento não encontrado").

O campo PROCOPT contém o valor PROCOPT especificado na instrução PCB, quando o PCB foi originalmente definido.

RESERVED é um campo reservado para uso do próprio IMS.

O campo SEGNAME contém o nome do último segmento que recebeu acesso.

O campo KEYFBLEN contém o comprimento significativo corrente da chave totalmente concatenada na área de recebimento da chave (veja KEYFBAREA abaixo).

O campo #SENSEGS contém uma contagem da quantidade de segmentos sensitivos. No nosso exemplo, o valor seria de 3 durante a execução.

O campo KEYFBAREA é a área de recebimento da chave; como explicado na Seção 17.2, esta área contém a chave totalmente concatenada (ajustada à esquerda) do último segmento que recebeu acesso.

O leitor poderá considerar que estes detalhes de programação têm pouco a ver com a abordagem hierárquica. No entanto, é necessário que se tenha um entendimento geral sobre este material para compreender a linguagem de manipulação de dados IMS (DL/I). As operações DL/I *podem* ser razoavelmente vistas como típicas da abordagem hierárquica, muito embora elas naturalmente envolvam uma certa quantidade de detalhes específicos do IMS.

18.2 AS OPERAÇÕES DL/I

A Fig. 18.2 resume as operações DL/I. Como explicado anteriormente, a aplicação invoca uma operação DL/I por meio de uma sub-rotina de chamada, na qual um dos parâmetros é o endereço do PCB apropriado. Os outros parâmetros incluem a operação requerida (por exemplo, GU), o endereço da área de entrada/saída, e (em alguns casos) uma ou mais condições de qualificação, conhecidas como "segment search arguments" (SSAs – argumentos de pesquisa do segmento). O nome da sub-rotina é fixada pelo IMS; para uma aplicação PL/I é PLITDLI. Para simplificar os exemplos, nós normalmente não usaremos a sintaxe DL/I genuína neste livro, mas sim uma sintaxe hipotética ilustrada pelo exemplo da Fig. 18.3, que mostra uma operação GU com três SSAs. (Um exemplo "genúino" encontra-se na Fig. 18.5.)

GET UNIQUE (GU)	Recuperação direta
GET NEXT (GN)	Recuperação seqüencial
GET NEXT WITHIN PARENT (GNP)	Recuperação seqüencial sob o pai corrente
GET HOLD (GHU, GHN, GHNP)	Como acima, mas permitindo subsequente DLET/REPL
INSERT (ISRT)	Adicione novo segmento
DELETE (DLET)	Remova segmento existente
REPLACE (REPL)	Substitua segmento existente

Fig. 18.2 Operações DL/I (Resumo).

Supondo que os dados sejam os mostrados na Fig. 16.2, o resultado da operação DL/I da Fig. 18.3 é o de recuperar o segmento STUDENT de "Gibbons, O." A explicação é como a que se segue. "Get unique" (GU) sempre provoca uma varredura seqüencial a partir do início do banco de dados¹ (pelo menos conceitualmente, pois sob o interface do usuário são normalmente usadas randomização ou indexação; veja o Capítulo 19). O segmento recuperado será o primeiro encontrado que satisfaça às três SSAs. As SSAs serão vistas com mais detalhes adiante; neste exemplo, as três SSAs especificam o caminho hierárquico até o segmento desejado — isto é, especificam o tipo de segmento em cada nível, da raiz para baixo, juntamente com uma condição de identificação da ocorrência para cada tipo. O efeito dessas SSAs é fazer com que o IMS pesquise avançando até encontrar a primeira ocorrência COURSE contendo 'DYNAMICS' como valor de título; de-

¹ Isto é verdade se tiver sido especificada uma SSA para a raiz. Veja [18.1] para detalhes do que acontece se não for este o caso.

```
GU COURSE  (TITLE='DYNAMICS')
OFFERING (FORMAT='F1' | FORMAT='F3')
STUDENT   (GRADE='A')
```

Fig. 18.3 Exemplo da sintaxe DL/I simplificada.

pois pesquise as ocorrências OFFERING subordinadas àquele COURSE até encontrar a primeira que contenha 'F1' ou 'F3' como valor de FORMAT; depois pesquise as ocorrências STUDENT subordinadas àquela OFFERING até encontrar a primeira contendo 'A' como valor de GRADE. Se não existir tal STUDENT naquela OFFERING, serão pesquisados os STUDENTS da próxima OFFERING com formato F1 ou F3 deste COURSE, e assim por diante. Caso não existam mais OFFERINGS com formato F1 ou F3 para este COURSE, o IMS irá pesquisar a próxima ocorrência de COURSE contendo 'DYNAMICS' como valor de TITLE, e repetir o processo (naturalmente pode não haver outro, e neste caso a operação de recuperação irá falhar, retornando um valor não-branco de status ao usuário).

Observe que só é retornado ao usuário o segmento que se encontra no fundo do caminho hierárquico. Observe também que em nossa sintaxe simplificada nós omitimos completamente a especificação da área de entrada/saída e do PCB (supusemos tacitamente que o banco de dados lógico cursos-ofertas-estudantes era o LDB relevante).

Em geral, uma SSA consiste de um nome de segmento, opcionalmente seguido por uma condição. Caso seja omitida a condição, qualquer ocorrência do segmento indicado irá satisfazer a esta SSA (desde que faça parte do caminho hierárquico definido pelas SSAs associadas). Se a condição for incluída, ela tem que consistir de um conjunto de expressões de comparação conectadas por meio dos operadores Booleanos "e" e "ou"; cada expressão de comparação consiste de uma tripla <campo, operador de comparação, valor>, na qual o campo tem que pertencer ao segmento especificado, e o operador de comparação pode ser qualquer um do conjunto usual (=, ≠, <, ≤, >, ≥). Todas as operações de comparação são executadas pelo IMS bit a bit da esquerda para a direita (isto é, não é assumida nenhuma representação particular dos dados para os valores envolvidos).²

Para simplificar um pouco as regras do IMS, podemos dizer que "get unique" e "insert" em suas operações requerem SSAs especificando todo o caminho hierárquico, desde a raiz; "get next" e "get next within parent" em suas operações podem ou não envolver SSAs, e, se o fizerem, essas SSAs terão novamente que especificar um caminho hierárquico, mas que pode ter início em qualquer nível hierárquico, não somente na raiz; e as operações "delete" e "replace" não envolvem SSAs. Mostraremos agora exemplos de todas essas possibilidades. Nesses exemplos consideraremos que o LDB é idêntico ao PDB do Capítulo 16 (isto é, todos os segmentos e todos os campos são sensitivos).

18.3 EXEMPLOS DL/I

Por conveniência, a estrutura de amostragem do banco de dados está novamente mostrada na Fig. 18.4.

² Com exceção do mencionado no Capítulo 22.

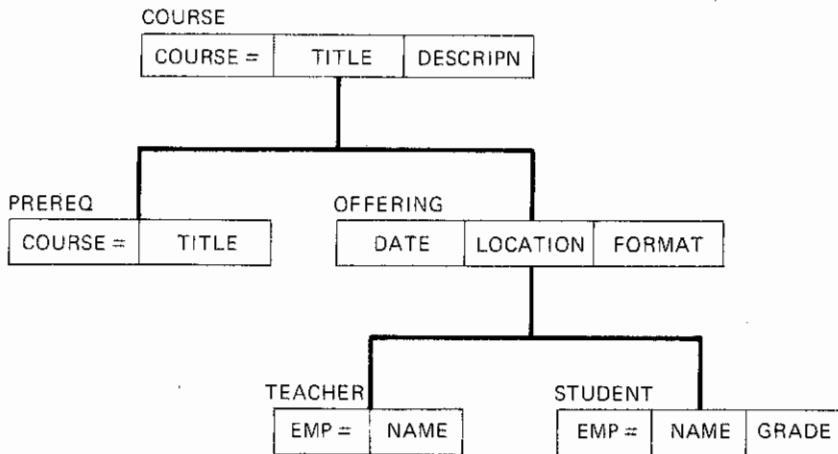


Fig. 18.4 Estrutura do banco de dados de educação.

18.3.1 Recuperação direta – Obtenha a primeira ocorrência OFFERING cuja localização seja Stockholm.

```
GU COURSE
OFFERING (LOCATION='STOCKHOLM')
```

Aqui estão ilustrados (a) o uso de uma SSA sem uma condição, e (b) um caminho hierárquico que termina antes do segmento de nível mais baixo. Incidentalmente, este “get unique”, bem como as outras operações DL/I devem na prática ser seguidos por um teste adequado do valor de status retornado; nós estaremos geralmente ignorando esta etapa nos nossos exemplos. Observe que “get unique” é uma designação algo incorreta — a operação é realmente “get first”.

18.3.2 Recuperação seqüencial com uma SSA – Obtenha todas as ocorrências STUDENT do LDB, começando pelo primeiro estudante da oferta encontrada no Exemplo 18.3.1.

```
GU COURSE
OFFERING (LOCATION='STOCKHOLM')
STUDENT
NS GN STUDENT
go to NS
```

O “get unique” recupera o primeiro estudante da primeira oferta em Stockholm. Isto estabelece uma posição corrente dentro do banco de dados. Na primeira vez em que for executado o “get next”, este irá recuperar o primeiro estudante que se segue a esta posição (no sentido de avanço), estabelecendo uma nova posição corrente; na segunda vez em que for executado, irá recuperar o próximo que se segue a *esta* posição, e assim por diante; todos os outros tipos de segmentos serão ignorados. Eventualmente será retornado um valor de status indicando “segmento não encontrado”.

Em geral, a operação de "get next" é definida em termos da posição corrente, sendo a posição corrente definida como o último segmento que recebeu acesso via uma operação "get" (de qualquer tipo) ou uma operação "insert".

18.3.3 Recuperação seqüencial com uma SSA condicional – Como no Exemplo 18.3.2, exceto que só deverão ser recuperadas as ocorrências STUDENT com grau A.

```
GU COURSE
OFFERING (LOCATION='STOCKHOLM')
STUDENT (GRADE='A')
NSA GN STUDENT (GRADE='A')
go to NSA
```

18.3.4 Recuperação seqüencial com uma SSA – Como no Exemplo 18.3.3, exceto que a pesquisa deve ser iniciada no princípio do banco de dados.

```
GU COURSE
NX GN STUDENT (GRADE='A')
go to NX
```

O "get unique" é usado aqui somente para estabelecer o primeiro segmento do banco de dados (a primeira ocorrência da raiz) como posição inicial. Observe que esse segmento também é recuperado, o que não é realmente necessário.

18.3.5 Recuperação seqüencial com múltiplas SSAs condicional – Como no Exemplo 18.3.3, exceto que só deverão ser recuperadas as ocorrências STUDENT com grau A das ofertas de Stockholm.

```
GU COURSE
OFFERING (LOCATION='STOCKHOLM')
STUDENT (GRADE='A')
NY GN OFFERING (LOCATION='STOCKHOLM')
STUDENT (GRADE='A')
go to NY
```

18.3.6 Recuperação seqüencial sem SSAs – Obtenha todos os segmentos.

```
GU COURSE
NZ GN
go to NZ
```

O usuário pode determinar o tipo de cada segmento após a sua recuperação inspecionando o PCB.

18.3.7 Recuperação seqüencial com uma SSA dentro de um pai – Obtenha todos os estudantes do curso M23 da oferta de 13 de agosto de 1973. (Veja a Fig. 16.2. Estamos supondo que haja apenas uma oferta desse curso nessa data.)

```
GU COURSE (COURSE#='M23')
OFFERING (DATE='730813')
NP GNP STUDENT
go to NP
```

A operação “get next within parent” é o mesmo que a operação “get next”, exceto que, depois de terem sido recuperados todos os segmentos satisfazendo às SSAs para o pai corrente, a próxima tentativa de se executar GNP irá indicar um valor de status indicando este fato. O pai corrente é o último segmento que recebeu acesso por meio de um “get unique” ou “get next” (*não* um “get next within parent”).

18.3.8 Recuperação seqüencial com uma SSA condicional dentro de um pai — Obtenha todos os estudantes que alcançaram o grau A no curso M23 (em qualquer oferta).

```
GU COURSE (COURSE#='M23')
NQ GNP STUDENT (GRADE='A')
go to NQ
```

Neste caso, o “pai” para a operação GNP não é o pai imediato do segmento a ser recuperado, mas sim seu “avô”. Qualquer segmento ancestral pode servir como “pai” para uma operação GNP.

18.3.9 Recuperação seqüencial sem SSAs dentro de um pai — Obtenha todos os segmentos subordinados do curso M23 (de todos os tipos).

```
GU COURSE (COURSE#='M23')
NN GNP
go to NN
```

Como no Exemplo 18.3.6, o usuário pode determinar o tipo de cada segmento após sua recuperação inspecionando o PCB.

18.3.10 Inserção de segmento — Adicione uma nova ocorrência STUDENT na oferta de 13 de agosto de 1973 do curso M23. (Estamos novamente supondo que só haja uma oferta desse curso nessa data.)

```
build new segment in I/O area
ISRT COURSE (COURSE#='M23')
OFFERING (DATE='730813')
STUDENT
```

Quando tem que ser inserida uma ocorrência de segmento, a ocorrência pai já tem que existir no banco de dados. A operação de “insert” especifica todo o caminho hierárquico até este pai (observe as condições nas SSAs de COURSE e OFFERING no exemplo) e também o tipo de segmento a ser inserido. O IMS dará entrada da ocorrência nova na posição correta, conforme definição do valor do seu campo de ordenação (neste caso EMP#).

De fato é possível omitir-se a especificação do caminho hierárquico e lançar-se somente o tipo do novo segmento. Neste caso, o IMS usará a posição corrente — isto é, o último segmento que recebeu acesso via operação “get” ou “insert” — para determinar onde inserir o novo segmento. Consideremos, por exemplo, a operação “insert”

ISRT STUDENT

Se o segmento corrente for um OFFERING, o novo STUDENT será inserido sob este; se for um TEACHER ou um STUDENT, será inserido sob o OFFERING que está sobre aqueles TEACHER ou STUDENT. Em qualquer caso, o valor do campo de ordenação será usado para a determinação da posição do novo segmento em relação a qualquer gêmeo existente.

“Insert” também é usado para se executar a carga inicial do banco de dados. Como as ocorrências de segmentos a serem carregadas têm que ser apresentadas em seqüência hierárquica,³ o método normal de operação neste caso é especificar-se somente o tipo do segmento na ISRT. (Basicamente cada novo segmento tem que ser carregado imediatamente após o segmento corrente.)

A ISRT coloca zeros binários em todos os campos não sensitivos do segmento PDB.

18.3.11 Remoção de segmento — Remova a oferta do curso M23 de 13 de agosto de 1973.

```
GHU COURSE (COURSE#='M23')
OFFERING (DATE='730813')
DLET
```

O segmento a ser removido tem que ser primeiramente recuperado via uma das operações “get hold” — “get hold unique” (GHU), “get hold next” (GHN), ou “get hold next within parent” (GHP). A operação “delete” pode então ser emitida (a menos que o usuário decida não remover o segmento depois de tudo, caso em que ele simplesmente continuará o processamento normalmente, por exemplo emitindo outro “get hold”). Observe que a operação “delete” não tem SSAs; entretanto, ela especifica a área de I/O (não está mostrado na nossa sintaxe simplificada), e o segmento recuperado permanece nessa área de I/O após a execução da remoção. Lembre-se de que uma operação de “delete” bem-sucedida remove o segmento especificado e também seus filhos todos.

18.3.12 Substituição de segmento — Mude a localização da oferta de 13 de agosto de 1973 do curso M23 para Helsinki.

```
GHU COURSE (COURSE#='M23')
OFFERING (DATE='730813')
mude a localização para "HELSINKI" na área de I/O REPL
```

³ Isto não é muito verdadeiro para HDAM; veja o Capítulo 19.

Como no caso de “delete”, o segmento a ser substituído tem que ser primeiramente recuperado via uma das operações “get hold”. Ele é então modificado na área de I/O, sendo então emitida a operação de “replace”. O campo de ordenação não pode ser modificado; seu valor tem que ser mantido sem alterações na área de I/O (incidentalmente, isto se aplica igualmente ao “delete”). De novo, se o usuário decidir não substituir o segmento, ele simplesmente continuará o processamento normalmente; e, de novo, não há SSAs envolvidas.

18.4 CONSTRUINDO O SEGMENT SEARCH ARGUMENT (SSA)

O processo de construção de argumentos de pesquisa de segmentos é um detalhe do IMS, não uma parte da abordagem hierárquica. No entanto, algo deve ser dito aqui sobre o assunto para evitar deixar uma falsa impressão no leitor. A sintaxe simplificada que viemos usando esconde o fato de que a SSA é na realidade uma seqüência de caracteres, formando um dos parâmetros de uma sub-rotina de chamada. Um valor típico desta seqüência de caracteres poderia ser.

```
'STUDENTb(GRADEbbb=bA)'
```

(Estamos agora mostrando a sintaxe IMS genuína, na qual têm que ser usados caracteres brancos de preenchimento — mostrados acima como b — para tornar cada porção da SSA um comprimento fixo predefinido, e em que o valor de comparação não é colocado entre apóstrofes). Neste exemplo o valor de comparação é uma constante, mas na prática é muito mais provável que seja requerido o valor de uma variável. Conseqüentemente, antes de emitir uma sub-rotina de chamada DL/I, o programador tem que *construir* dinamicamente a seqüência de caracteres SSA movendo o valor da variável de comparação para a posição apropriada. (Naturalmente, em geral, o programador poderá alterar dinamicamente qualquer porção da SSA desta forma.) Como uma ilustração, estamos mostrando na Fig. 18.5 parte de um programa PL/I usando sintaxe IMS genuína. O código mostrado corresponde à primeira chamada DL/I (GU) do Exemplo 18.3.3.

18.5 CÓDIGOS DE COMANDO SSA

Uma SSA pode opcionalmente incluir um ou mais “códigos de comando”. Cada código de comando é representado por um caractere singelo (por exemplo, F para “first”); os códigos de comando são especificados escrevendo-se um asterisco seguido pelo(s) caractere(s) apropriado(s) após o nome do segmento na SSA. Daremos exemplos do uso dos códigos de comando D (provavelmente o mais útil do conjunto), F, e V. Observe que no caso do código de comando D, o PROCOPT no PCB tem que incluir a entrada P [16.1]. Para detalhes sobre os códigos restantes (C, L, P, Q, U, N, —), veja a referência [18.1].

18.5.1 Recuperação de caminho Obtenha a primeira ocorrência de OFFERING em que a localização seja Stockholm, juntamente com sua ocorrência pai COURSE (como o Exemplo 18.3.1).

```
GU  COURSE*D  
OFFERING (LOCATION='STOCKHOLM')
```

O “D” significa dados. O efeito deste “get unique” é o de localizar a primeira oferta em Stockholm (como no exemplo 18.3.1), e depois recuperar todo o caminho hierárquico de

```

DLITPLI: PROC(EDPCB_ADDR) OPTIONS(MAIN);

DCL 1 EDPCB BASED(EDPCB_ADDR), . . . ;
DCL STUDENT_AREA CHAR(25); /* input area */

DCL 1 CSSA,
  2 CSEGNM CHAR(8) INITIAL('COURSEbb'),
  2 CSSAEND CHAR(1) INITIAL('b');

DCL 1 OSSA,
  2 OSEGNAME CHAR(8) INITIAL('OFFERING'),
  2 OLPAREN CHAR(1) INITIAL('('),
  2 OFLDNAME CHAR(8) INITIAL('LOCATION'),
  2 OCOMPPOP CHAR(2) INITIAL('=b'),
  2 OFLDVAL CHAR(12),
  2 ORPAREN CHAR(1) INITIAL(')');

DCL 1 SSSA,
  2 SSEGNM CHAR(8) INITIAL('STUDENTb'),
  2 SLPPAREN CHAR(1) INITIAL('('),
  2 SFLDNAME CHAR(8) INITIAL('GRADEbbb'),
  2 SCOMPPOP CHAR(2) INITIAL('=b'),
  2 SFLDVAL CHAR(1),
  2 SRPAREN CHAR(1) INITIAL(')');

DCL GU CHAR(4) INITIAL('GUbb');
DCL SIX FIXED BIN(31) INITIAL(6);

*****
      OFLDVAL='STOCKHOLMbbb'; /* na prática, seriam variáveis,
      SFLDVAL='A'; /* não constantes
      CALL PLITDLI (SIX, GU, EDPCB, STUDENT_AREA, /* call DL/I */
                    CSSA, OSSA, SSSA);
      IF EDPCB.STATUS='GE' THEN.../* segmento não encontrado */
*****
END DLITPLI;

```

Fig. 18.5 Exemplo de sintaxe IMS.

segmentos até aquele ponto — neste caso, dois segmentos. Em geral, o código de comando D pode ser especificado em alguns níveis e não em outros; o efeito é o de recuperar somente os segmentos indicados no caminho hierárquico e concatená-los na área de I/O. Observe que não é necessário especificar-se *D no nível mais baixo do caminho, pois este segmento será recuperado de qualquer maneira.

O leitor deve estar imaginando como este exemplo poderia ser manuseado sem o uso do caminho de recuperação. (Uma possibilidade: recuperar a ocorrência de OFFERING como no Exemplo 18.3.1; extrair o número de COURSE da área de recebimento da chave no PCB; usando este valor, construir uma SSA e então recuperar diretamente o COURSE, via outro “get unique”.)

Se o “get hold” que precede uma operação “delete” ou “replace” for de recuperação de caminho, a operação DLET/REPL será normalmente considerada como se aplicando a todo o caminho. No entanto, é possível ser mais seletivo do que isto; para detalhes veja a referência [18.1].

18.5.2 Inserção de caminho — Insira um novo curso (M40), juntamente com uma oferta em Brussels em 4 de janeiro de 1976, na qual o professor será o empregado número 876225.

construa os três segmentos concatenados na área I/O

```
ISRT COURSE*D  
OFFERING  
TEACHER
```

Observe que só é requerido *D para o primeiro segmento do caminho. Este segmento é inserido (no exemplo é uma raiz, e por isso não tem pai); o segundo segmento no caminho pode agora ser inserido, pois agora seu pai existe; e semelhantemente para o terceiro.

18.5.3 Uso do código de comando F — Obtenha o professor (supondo que só haja um) da primeira oferta (de qualquer curso) da qual participou o estudante 183009.

```
GU COURSE  
NO GN OFFERING  
GNP STUDENT (EMP#='183009')  
se não for encontrado vá para NO  
GNP TEACHER*F
```

O “get unique” nos posiciona no início do banco de dados. O “get next” estabelece uma oferta como o pai corrente, e o “get next within parent” de STUDENT pesquisa para verificar se o estudante especificado participou desta oferta. Estas duas operações são repetidas até que seja encontrada uma oferta da qual este estudante tenha participado. Agora nós desejamos recuperar o professor correspondente. Entretanto, o “get next within parent” irá retornar (sem o *F) “segmento não encontrado”, pois pesquisará no sentido de avanço, e professor precede os estudantes na seqüência hierárquica. De forma semelhante, “get next” (não o dentro do pai) irá recuperar um professor, mas será o professor de alguma oferta subsequente. O que nós precisamos é de um meio de *voltar atrás* sob o pai corrente, e é isto que o *F faz; ele faz com que o IMS inicie a pesquisa na primeira ocorrência do tipo de segmento especificado sob o pai corrente. Neste exemplo, como a SSA de TEACHER é incondicional, a primeira ocorrência é de fato a que satisfaz à pesquisa, sendo por isso recuperada.

18.5.4 Uso do código de comando V — Obtenha todos os estudantes do curso M23 da oferta de 13 de agosto de 1973 (mesmo do exemplo 18.3.7).

```
GU COURSE (COURSE#='M23')  
OFFERING (DATE='730813')  
NP GN OFFERING*V  
STUDENT  
go to NP
```

Esta seqüência de operações produz exatamente o mesmo resultado que no Exemplo 18.3.7. Antes de uma explicação detalhada, é necessária uma ampliação do conceito de “posição corrente”. Basicamente, posição corrente é definida como sendo o último segmento que recebeu acesso via uma operação “get” ou “insert”. No entanto, em adição, cada *ancestral* do segmento corrente — isto é, cada segmento no caminho entre o segmento corrente a a raiz — é considerado como o corrente do tipo de segmento relevante. Por

exemplo, se o último segmento recuperado tiver sido STUDENT, então aquele STUDENT é o segmento corrente; a OFFERING pai daquele STUDENT também é corrente, e o pai daquela OFFERING é o COURSE corrente. Esta OFFERING e este COURSE são os segmentos correntes de seus respectivos tipos, mas não são o segmento corrente.

Uma SSA com código de comando V faz com que o IMS não saia do segmento corrente do tipo cujo nome está na SSA ao pesquisar um segmento que satisfaça à solicitação. Voltando ao nosso exemplo, seja X a OFFERING corrente localizada pelo GU. A operação GN pode então ser parafraseada como “Obtenha o próximo STUDENT sob X” (devido ao *V na SSA de OFFERING). Em outras palavras, é equivalente a um “get next within parent” para os STUDENTS sob X. Assim, o código acima tem o mesmo efeito que o do Exemplo 18.3.7.

Este exemplo não ilustra o verdadeiro valor de *V. A vantagem de se usar *V preferivelmente no lugar de GNP é que o “pai” referenciado por *V não tem que ser o “pai corrente” como definido no Exemplo 18.3.7 – ao invés, pode ser qualquer ancestral do segmento corrente. O exemplo a seguir demonstra este ponto.

18.5.5 Uso do código de comando V – Obtenha o professor da primeira oferta da qual participou o estudante 183009 (mesmo do Exemplo 18.5.3).

```
GU STUDENT (EMP#='183009')
GN OFFERING*V
      TEACHER*F
```

O GU recupera a primeira ocorrência de STUDENT do estudante 183009 no banco de dados. Esta recuperação traz como efeito (entre outras coisas) tornar OFFERING corrente o pai deste STUDENT. O GN então recupera o TEACHER sob esta OFFERING (observe que ainda é necessário o *F em TEACHER).

Comparando este exemplo com o Exemplo 18.5.3, vemos que (a) não precisamos agora de um GU especial para posicionar-nos no início do banco de dados, (b) nem precisamos recuperar OFFERINGS – tivemos que fazer isto antes simplesmente para estabelecer o “pai corrente” – e finalmente (c) eliminamos um *loop*, pois agora podemos ir diretamente à ocorrência desejada de STUDENT.

O código de comando V não pode ser usado no nível mais baixo de uma seqüência de SSAs, nem pode ser usado em uma SSA que inclua condições de qualificação dos campos do segmento. Entretanto, como ele fornece quase todas as funções do GNP e outras mais, deverá sempre que possível ser usado no lugar do GNP.

18.6 USANDO MAIS DE UM PCB

Consideremos o problema “Dado um número de curso (digamos M23), imprima um relatório contendo todas as ofertas de todos os cursos pré-requisito daquele curso dado”. Um esboço de solução para este problema encontra-se a seguir:

```
DO until no more PREREQs for given COURSE
    get next PREREQ for given COURSE
    get COURSE where COURSE# = course# of PREREQ
    DO until no more OFFERINGS for latter COURSE
        get next OFFERING for latter COURSE
        print OFFERING
    END
END
```

Fica claro que esta lógica requer a capacidade de se manter simultaneamente duas posições no banco de dados – o *loop* externo baseia-se em uma posição que se move através dos PREREQs de um determinado curso; o *loop* interno baseia-se em uma segunda posição que se move através das OFFERINGS do curso que corresponde ao PREREQ identificado pela primeira posição. A primeira posição não pode se alterar enquanto é executado o *loop* interno.

Como pode esse problema ser programado em DL/I? A resposta é pelo uso de dois PCBs. Lembre-se de que cada operação DL/I tem como um dos seus parâmetros o endereço de um PCB (veja um exemplo na Fig. 18.5). O PCB especificado identifica o banco de dados apropriado ao IMS e também uma posição corrente dentro daquele banco de dados. Isto é, o IMS guarda a “posição corrente” (veja o Exemplo 18.3.2) gravando-a em um campo “escondido” do PCB (isto é, um campo ao qual o usuário não tem acesso), e operações tais como GN fazem uso implícito deste campo escondido. Para o problema em questão, portanto, o PSB da aplicação deverá incluir dois PCBs, ambos correspondendo ao banco de dados de educação. Chamemos estes dois PCBs de PPCB e OPCB. O problema poderá então ser codificado como se segue (ainda usando a sintaxe simplificada):

```
GU COURSE(COURSE# = given course#) [USING PPCB]
DO until 'not found' on PPCB
    GN COURSE*V
        PREREQ [USING PPCB]
    GU COURSE(COURSE# = PREREQ course#) [USING OPCB]
    DO until 'not found' on OPCB
        GN COURSE*V
            OFFERING [USING OPCB]
            print OFFERING
    END
END
```

Podemos ver agora que há semelhanças entre os PCBs da DL/I e os cursores da SQL embutida (embora o PCB leve muito mais informações do que o cursor).

EXERCÍCIOS

Os exercícios a seguir estão baseados no banco de dados de publicações (veja o Exercício 16.2). Podemos supor que todos os segmentos e campos são sensitivos. Ignore o teste do valor de status tanto quanto possível.

- 18.1 Obtenha todos os autores de publicações sobre o assunto “recuperação de informações” (podemos supor que este nome de assunto seja único).
- 18.2 Obtenha os nomes de todas as publicações das quais são autores Grace ou Hobbs (um deles).
- 18.3 Obtenha os nomes de todos os assuntos sobre os quais Bradbury publicou monografias.
- 18.4 Obtenha o nome e a data da primeira publicação de todos os artigos publicados por Owen.
- 18.5 Obtenha os nomes de todos os autores que tiveram monografias publicadas pela Cider Press desde 1 de janeiro de 1970.
- 18.6 Uma monografia sobre ficção científica, chamada “Computers in SF”, foi publicada em 1 de janeiro de 1973 pela Galactic Publishing Corporation. O nome do autor é Hal. Adicione esta informação ao banco de dados. (Podemos supor que o assunto “ficção científica” já esteja representado.)

18.7 De todas as publicações correntemente disponíveis em mais de uma fonte, remova todos os segmentos de detalhes exceto o mais recente. (Pode ser ignorada a possibilidade de que dois segmentos possam incluir os mesmos dados, sendo consequentemente ambos "mais recentes".)

18.8 Para cada uma das questões anteriores (onde for aplicável), determine se o uso de *V ao invés de GNP oferece alguma vantagem.

REFERÊNCIAS E BIBLIOGRAFIA

18.1 IBM Corporation. Information Management System/Virtual Storage Application Programming Reference Manual. IBM Form n° SH20-9026.

19

O Nível Interno do IMS

19.1 INTRODUÇÃO

Continuando nosso estudo do IMS como um exemplo da abordagem hierárquica, vamos agora observar o que ocorre sob o interface do usuário e a estrutura de memória do IMS. O IMS fornece quatro diferentes estruturas de memória, conhecidas como HSAM, HISAM, HDAM, e HIDAM,¹ e um banco de dados armazenado (isto é, a representação armazenada de um banco de dados físico) pode estar em qualquer das quatro formas.

No Capítulo 2 nós fizemos uma distinção entre o DBMS por um lado, e o método de acesso por ele usado por outro lado. A função do método de acesso é o de apresentar ao DBMS um “interface do registro armazenado”. Este interface de fato existe dentro de um sistema IMS, embora o quadro seja bem mais complicado do que o apresentado no Capítulo 2. Veja a Fig. 19.1.

O primeiro ponto é que o IMS usa diversos métodos de acesso sob o interface do registro armazenado, não apenas um. São:

- OS/VS Sequential Access Methods (QSAM e BSAM, coletivamente conhecidos como SAM);
- OS/VS Indexed Sequential Access Method (ISAM);
- OS/VS Virtual Storage Access Method (VSAM);
- Um método de acesso especial do IMS chamado Overflow Sequential Access Method (OSAM).

Por exemplo, as rotinas do HISAM — as rotinas do programa de controle do IMS que processam os bancos de dados HISAM — usam ou o VSAM ou uma combinação de

¹ Estamos seguindo aqui o uso padrão do IMS, embora estritamente falando fosse mais preciso reservar esses nomes para os conjuntos correspondentes de rotinas dentro do programa de controle do IMS; veja a explicação sobre as siglas a seguir.

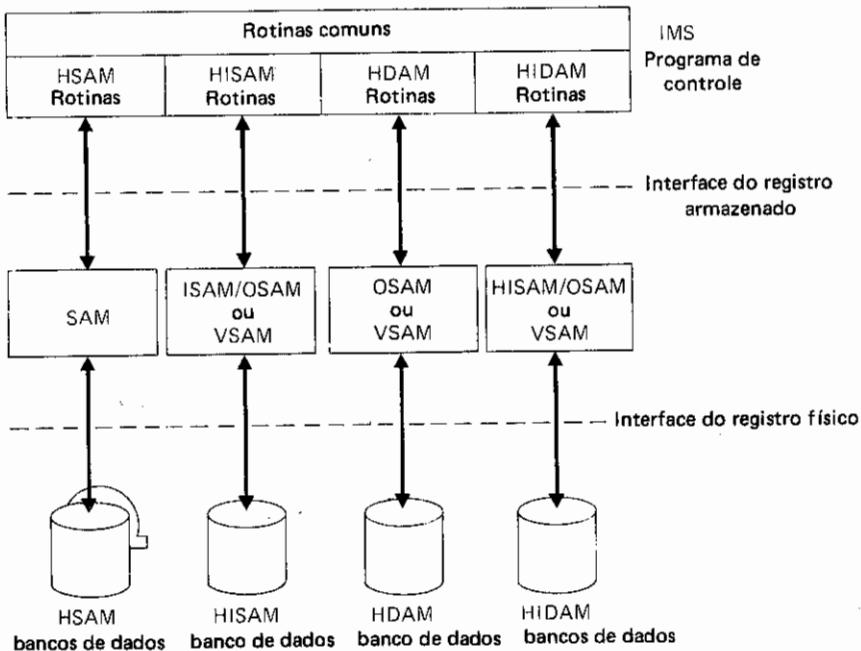


Fig. 19.1 Métodos de acesso e estruturas de memória do IMS.

ISAM o OSAM, e a função desses métodos de acesso é fazer com que as rotinas do HISAM considerem o banco de dados básico como estando na estrutura HISAM. Portanto, as rotinas do HISAM não se envolvem com detalhes (por exemplo) da indexação requerida para suporte a esta estrutura específica. Não é nosso objetivo aqui descrever os métodos de acesso em detalhes sob o interface do registro armazenado; essas descrições podem ser encontradas em [19.2] (para SAM e ISAM), [19.4] (para VSAM), e [19.1] (para OSAM).

O termo “método de acesso” também é usado para as rotinas do programa de controle IMS que processam as quatro diferentes estruturas. Esses “métodos de acesso” são como se segue:²

- Hierarchical Sequential Access Method (HSAM – método de acesso seqüencial hierárquico);
- Hierarchical Indexed Sequential Access Method (HISAM – método de acesso seqüencial indexado hierárquico);

² Em adição a estes quatro, o IMS também fornece um método de acesso conhecido como GSAM (Generalized Sequential Access Method), que permite o uso de um conjunto muito restrito de operadores DL/I sobre arquivos seqüenciais comuns OS/VS (SAM ou VSAM). O uso do GSAM poderia significar que o programa não se envolve na entrada/saída convencional – todas essas operações seriam manuseadas via DL/I. No entanto, o GSAM não é realmente um método de acesso a *bancos de dados*, e nós iremos ignorá-lo a partir de agora.

- Hierarchical Direct Access Method (HDAM – método de acesso direto hierárquico);
- Hierarchical Indexed Direct Access Method (HIDAM – método de acesso direto indexado hierárquico).

A situação complica-se um pouco pelo fato de as rotinas do HISAM também aparecerem sob o interface do registro armazenado como um dos métodos de acesso usados para o HIDAM. No entanto isto realmente não afeta o quadro geral. A seguir nós estaremos geralmente nos restringindo à discussão ao nível do interface do registro armazenado.

Como já dissemos, cada banco de dados físico (PDB) é representado como um banco de dados armazenado em uma das quatro estruturas. Dentro do banco de dados armazenado, cada ocorrência de segmento PDB é representada por meio de uma ocorrência de segmento *armazenado*, que contém os dados – exatamente como o usuário os vê – juntamente com um *prefixo* que o usuário não vê.⁴ O prefixo contém informações de controle do segmento: indicadores de remoção, código do tipo de segmento, indicadores de localização e assim por diante. Portanto, os segmentos PDB são representados essencialmente da mesma maneira em cada uma das quatro estruturas. Onde as quatro estruturas diferem é na forma usada para (a) agrupar as ocorrências PDB para formar ocorrências PDBR, e (b) agrupar as ocorrências PDBR para formar o PDB. Em outras palavras, a diferença reside na maneira pela qual é representada a seqüência hierárquica do PDB.

Vamos agora prosseguir, investigando cada uma das quatro estruturas com maiores detalhes.

19.2 HSAM

O que melhor descreve o HSAM é *semelhante a fita*. Como podemos ver na Fig. 19.1, o banco de dados HSAM pode residir em uma fita. A seqüência hierárquica é inteiramente representada pela configüidade física. Veja, por exemplo, a Fig. 19.2 (onde estamos supondo que o curso M27 seja o próximo na seqüência após o curso M23 no banco de dados de educação – não há M24, M25, ou M26).

Assim, um banco de dados HSAM é representado por meio de um único *arquivo* [19.2] contendo registros armazenados de comprimento fixo.⁵

Cada registro armazenado pode conter qualquer quantidade de segmentos armazenados; entretanto, cada segmento armazenado tem que estar inteiramente contido em um registro armazenado (isto é, segmentos armazenados não podem se “estender” por vários registros armazenados), o que significa que podem ficar sem uso *bytes* no final de um registro armazenado – dependendo do comprimento do registro armazenado escolhido pelo DBA. (De fato, nenhuma das quatro estruturas permite que segmentos se estendam por vários registros armazenados.) Entretanto, deixando de lado os espaços vazios ocasionais que isto possa causar, cada registro armazenado encontra-se imediatamente seguido pelo seu sucessor na seqüência hierárquica.

³ Nos casos de HDAM e HIDAM, segmentos PDB que consistam de um número ímpar de *bytes* serão estendidos para um número par pela adição de um *byte* de preenchimento, não visível ao usuário.

⁴ Segmentos em “HSAM simples” e “HISAM simples” não têm um prefixo. Veja a Seção 19.8.

⁵ O termo OS/VS para registro armazenado é “registro lógico”.

COURSE M23	PREREQ M16	PREREQ M19	OFFERING 730813	TEACHER 421633	STUDENT 102141	STUDENT 183009	...
STUDENT 761620	OFFERING 741104	OFFERING 750106	COURSE M27	PREREQ L02	OFFERING 740602	TEACHER 421633	TEACHER 502417

Fig. 19.2 Parte do banco de dados de educação (HSAM).

O banco de dados HSAM é criado por meio de uma série de operações “insert”; observe que os segmentos a serem carregados têm que ser apresentados em seqüência hierárquica. Uma vez criado, o banco de dados só pode ser usado como entrada; isto é, só são válidas operações “get” (GU, GN, e GNP; não as operações “get hold”). Assim, o uso mais comum do HSAM envolve a técnica familiar de “mestre anterior/novo mestre” do processamento de arquivos seqüenciais. Em outras palavras, as atualizações são aplicadas a uma versão existente do banco de dados para gerar uma nova versão (fisicamente separada). As operações DL/I “delete” e “replace” não podem ser usadas. Ao invés, o usuário pode remover um segmento existente não o inserindo no novo banco de dados, e pode substituir um segmento modificando-o antes de inseri-lo no novo banco de dados.

19.3 HISAM

Podemos caracterizar o HISAM, de uma forma algo imprecisa, dizendo que este fornece acesso indexado aos segmentos raiz, e acesso seqüencial aos segmentos subordinados. (O índice está no campo de ordenação do segmento raiz.) Esta afirmativa é verdadeira para o banco de dados HISAM pelo menos logo após este ter sido carregado; entretanto, como veremos, o quadro pode se tornar algo distorcido após terem ocorrido inserções e remoções. O quadro também é um pouco diferente se forem usados grupos de arquivos secundários; vamos deixar a discussão deste caso para a Seção 19.7.

Como mencionamos na Seção 19.1, o HISAM usa VSAM ou uma combinação de ISAM e OSAM como método de acesso de suporte. Vamos examinar primeiro o caso ISAM/OSAM, e depois descrever as diferenças para o VSAM.

HISAM usando ISAM/OSAM

Um banco de dados HISAM sob ISAM/OSAM consiste de dois arquivos, um arquivo ISAM e um arquivo OSAM. Cada um deles encontra-se dividido em registros armazenados de comprimento fixo. Em ambos, os registros armazenados são criados seqüencialmente e recuperados seqüencial ou diretamente — do ISAM via o índice ISAM (geralmente não-denso, incidentalmente; veja a Seção 2.3), e do OSAM via o endereço relativo do registro dentro do arquivo (comparável ao SRA da Seção 2.1). A Fig. 19.3 mostra a estrutura de um desses registros armazenados.

Quando o banco de dados é inicialmente carregado (lembre-se de que a carga tem que se feita na seqüência hierárquica), cada segmento raiz apresentado provoca a criação de um novo registro armazenado ISAM; o segmento raiz é colocado na frente deste registro e após esta raiz serão colocados tantos segmentos dependentes quantos caibam. Se o registro ISAM for preenchido antes que todos os dependentes da raiz corrente te-

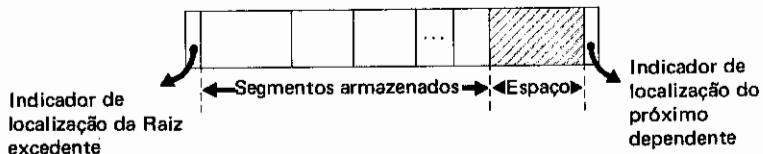


Fig. 19.3 Estrutura de um registro armazenado em HISAM (usando ISAM/OSAM).

nham sido carregados, então é criado um novo registro OSAM, e o próximo segmento dependente é colocado na frente deste; ao mesmo tempo o endereço relativo do registro OSAM é colocado na posição “indicador de localização do próximo dependente” do registro ISAM.⁶ Os dependentes subsequentes seguem-se a este no registro OSAM. Se, por sua vez, o registro OSAM for preenchido, é criado outro (sendo o seu endereço colocado na anterior), e assim por diante. Assim, cada ocorrência de registro do banco de dados físico forma uma cadeia contendo um registro ISAM juntamente com zero ou mais registros OSAM; qualquer espaço livre no final de um registro da cadeia é considerado como pertencente à ocorrência PDBR envolvida (poderá ajudar em operações subsequentes de “insert”).

A Fig. 19.4 mostra como parte do banco de dados de educação poderia aparecer em HISAM logo após a carga.

A remoção de um segmento em HISAM é feita pela colocação de um indicador no prefixo do segmento. Os dependentes de um segmento removido são automaticamente considerados como removidos; normalmente não é necessária a colocação nestes de indicadores de remoção, pois qualquer tentativa de acesso a eles tem que ser, de qualquer forma, via o segmento removido. (Isto *não* é necessariamente verdadeiro para um banco de dados HDAM ou HIDAM; veja o Capítulo 20.) O segmento removido continua a ocupar espaço no banco de dados; este espaço não fica disponível para novo uso.

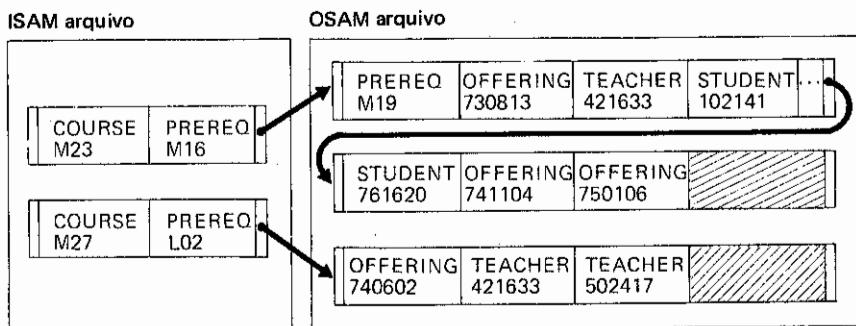


Fig. 19.4 Parte do banco de dados de educação (HISAM, usando ISAM/OSAM).

⁶

Na prática, o último segmento armazenado no registro é seguido por um *byte* de zeros (efetivamente um segmento com código de tipo zero), e o indicador de localização do “próximo dependente” segue-se imediatamente a este *byte*. A Fig. 19.3 e outras mostram, para maior clareza, o indicador de localização na extremidade do registro armazenado.

Para a inserção, o método de operação depende de ser o novo segmento uma raiz ou um subordinado. Se for uma raiz, é criado um novo registro OSAM e o novo segmento colocado na frente deste. Seja Y o valor do campo de ordenação da nova raiz, e sejam X e Z os valores dos campos de ordenação das raízes que imediatamente precedem e sucedem a nova raiz, respectivamente, na seqüência hierárquica.⁷ Suponhamos que as raízes X e Z estão no (registros armazenados consecutivos) arquivo ISAM; isto é, esta foi a primeira inserção ocorrida nesta posição. Então, como explicado acima, a raiz Y é colocada no novo registro OSAM; também um indicador de localização deste registro OSAM é colocado no “indicador de localização da raiz excedente” do registro ISAM que contém a raiz Z. Se agora for inserida uma nova raiz, com o valor Y' do campo de ordenação tal que $X < Y' < Z$, novamente será criado um novo registro OSAM. Se $Y < Y'$, será colocado no registro Y um indicador de localização do novo registro OSAM; mas se $Y' < Y$, então o indicador de localização do registro ISAM será modificado para indicar o registro Y', e será colocado um indicador de localização no registro Y' para indicar o registro Y. Em geral, o registro Z poderá indicar uma cadeia com qualquer quantidade de registro OSAM, cada um contendo um segmento raiz inserido, e esta cadeia será mantida em ordem ascendente de seqüência de raiz. A Fig. 19.5 mostra a situação depois de terem sido inseridas (a) a raiz M26 e depois (b) a raiz M24, nessa ordem, no banco de dados HISAM da Fig. 19.4 (só estão mostrados os segmentos raiz). Observe, incidentalmente, que como resultado da técnica usada para inserção de raiz, nunca é entrada uma nova chave no índice ISAM após a carga do banco de dados.

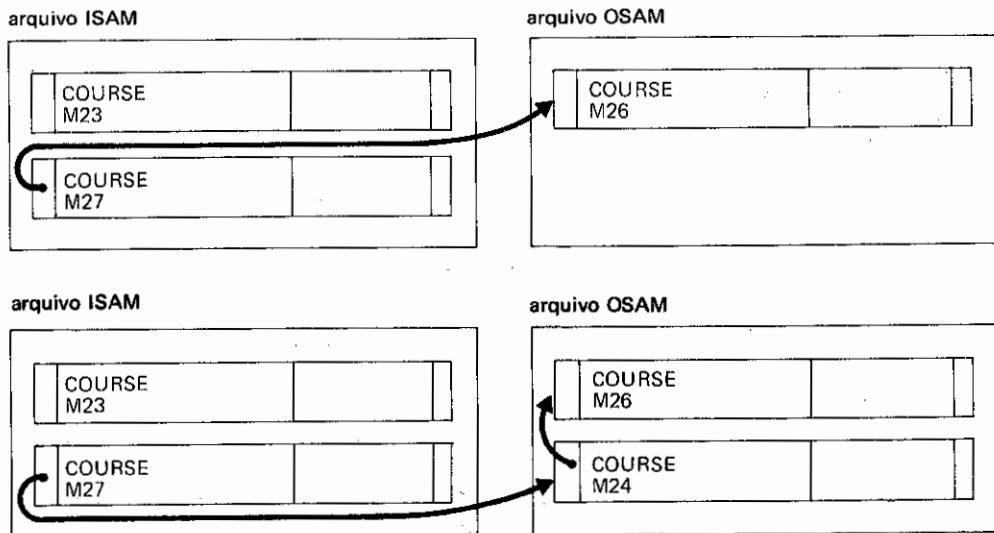


Fig. 19.5 Exemplos de inserção de segmentos raiz (HISAM, usando ISAM/OSAM).

⁷

O segmento Z sempre existe, pois o IMS coloca automaticamente uma raiz fictícia no banco de dados ao final do processo de carga, com um valor de campo de ordenação maior do que qualquer valor real possível. O segmento X pode não existir, mas isto realmente não importa, como a explicação a seguir esclarece.

Os segmentos dependentes são inseridos no local correto em seqüência hierárquica. Isto requer uma varredura na cadeia dos registros armazenados que representam a ocorrência PDBR, para encontrar o predecessor do novo segmento. Os segmentos que se seguem ao predecessor (se existirem) serão deslocados para a direita dentro do registro armazenado para darem espaço ao novo segmento.⁸ Se houver espaço livre suficiente para acomodar o novo segmento e os segmentos deslocados, não será necessário processamento adicional. Entretanto, é frequente o caso em que um ou mais segmentos não mais cabem no segmento; pode não haver sequer espaço para o próprio novo segmento (esta situação ocorre quando o comprimento excede o número de bytes entre o predecessor e o final do registro). Todos esses segmentos excedentes são colocados no arquivo OSAM, ocupando um ou possivelmente dois registros OSAM. O indicador de localização do "próximo dependente" na cadeia é adequadamente ajustado para manter a seqüência exigida. A Fig. 19.6 ilustra a situação após a inserção de um segmento PREREQ (L01) subordinado à raiz M27 (só estão mostrados registros armazenados relevantes).

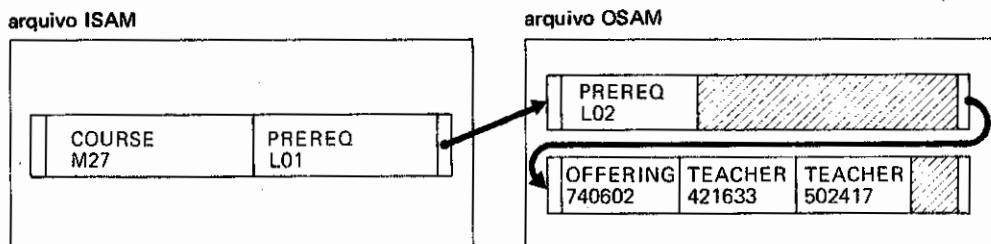


Fig. 19.6 Exemplo de inserção de segmento dependente (HISAM, usando ISAM/OSAM).

HISAM usando VSAM

Estamos resumindo abaixo os pontos mais importantes nos quais o HISAM usando VSAM difere do HISAM usando ISAM/OSAM.

- Os arquivos ISAM e OSAM são substituídos respectivamente por um arquivo VSAM por seqüência de chave e um arquivo VSAM por seqüência de entrada.
- Todos os segmentos raiz residem no arquivo por seqüência de chave, mesmo as inserções. Quando é inserida uma nova raiz, ela é colocada em sua posição apropriada neste arquivo, sendo o índice do VSAM atualizado se necessário. O indicador de localização da raiz excedente mostrado na Fig. 19.3 para ISAM/OSAM não existe para VSAM. (Na verdade, o indicador de localização do "próximo dependente" aparece em seu lugar, isto é, na frente do registro, ao invés de após todos os segmentos armazenados no registro como no ISAM/OSAM.)
- Sob certas circunstâncias — veja [19.1] para detalhes —, a remoção de um segmento raiz irá liberar o espaço de armazenamento do registro que o contém (no arquivo por seqüência de chave) para uso futuro.

⁸

Os segmentos removidos são tratados como os outros segmentos neste processo.

Concluindo nossa discussão sobre HISAM, vamos ampliar as observações feitas no início da seção. O acesso aos segmentos raiz em um banco de dados HISAM é feito por meio de um índice (ISAM ou VSAM) no campo de ordenação do segmento raiz. Entretanto, em geral, a indexação é somente parcial pelas seguintes razões:

- O índice é não-denso, havendo em geral muitas raízes em cada entrada de índice — basicamente N por entrada, onde N é o número de registros ISAM por trilha ou registros VSAM por intervalo de controle.⁹
- Além disso, podem existir algumas raízes (inserções) no arquivo OSAM ao invés de no arquivo ISAM se estiver sendo usada a combinação ISAM/OSAM.

O acesso a segmentos dependentes é seqüencial, da mesma forma que para se ter acesso ao N -ésimo dependente (em seqüência hierárquica) de uma determinada raiz é necessário atravessar-se os $(N-1)$ dependentes que o precedem. Isto pode envolver a passagem por segmentos removidos. Pode também envolver a travessia de cadeias de um registro armazenado para outro.

19.4 ESTRUTURAS HD: INDICADORES DE LOCALIZAÇÃO

As duas estruturas diretas hierárquicas, HIDAM e HDAM, envolvem o uso de indicadores de localização para unir os segmentos. Se o método de acesso básico for VSAM, todos os dados no banco de dados são armazenados em um ou mais arquivos VSAM por seqüência de entrada; caso contrário, são armazenados em um ou mais arquivos OSAM. Em qualquer dos casos, os indicadores de localização mencionados consistem de *bytes* de deslocamentos dentro do arquivo relevante. Fisicamente eles são armazenados como parte do prefixo do segmento. São usados para (a) representar a seqüência hierárquica dos segmentos dentro de uma ocorrência PDBR, e (b) para representar a seqüência das ocorrências PDBR — pelo menos até certo ponto. Vamos analisar esses indicadores de localização com alguns detalhes antes de discutirmos HDAM e HIDAM.

Consideremos inicialmente a seqüência hierárquica de segmentos dentro de uma ocorrência PDBR. Isto pode ser basicamente representado de duas formas, por meio de indicadores de localização “hierárquicos” ou por meio de indicadores de localização “filho/gêmeo”. A Fig. 19.7 mostra uma ocorrência PDBR na qual são usados indicadores de localização hierárquicos.

Observe que dentro da ocorrência PDBR armazenada, cada segmento inclui um indicador da localização do próximo na seqüência hierárquica. (Entretanto, o último segmento não inclui um indicador de localização da próxima raiz.) Como opção, os indicadores de localização hierárquicos podem ser estabelecidos nos dois sentidos, isto é: cada segmento pode adicionalmente incluir um indicador de localização do seu predecessor (novamente, dentro de uma ocorrência PDBR).

A Fig. 19.8 mostra como a mesma ocorrência PDBR ficaria se fossem usados indicadores de localização filho/gêmeo. Ali, cada ocorrência de segmento pai inclui um indicador da localização da primeira ocorrência de cada um dos tipos de seus segmentos filhos, e cada ocorrência de segmento filho inclui um indicador da localização da próxima ocorrência — se houver — daquele filho sob o pai corrente (isto é, do próximo gêmeo). Tal co-

⁹ No entanto, o DBA pode fazer com que $N = 1$.

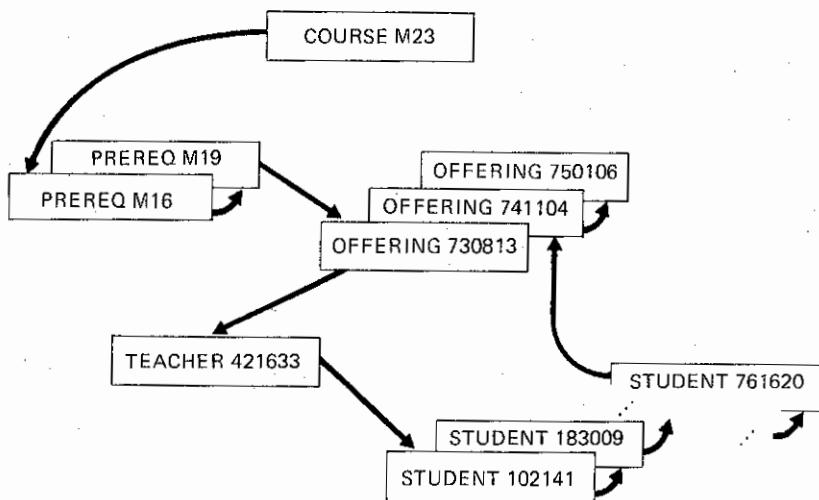


Fig. 19.7 Exemplo de indicadores de localização hierárquicos.

mo para os indicadores de localização hierárquicos, esses indicadores de localização gêmeos podem opcionalmente ser estabelecidos nos dois sentidos. Além disso, cada pai pode incluir um indicador da localização da última ocorrência (bem como da primeira) de qualquer ou de todos tipos dos seus segmentos filhos.

Vamos agora analisar a seqüência de ocorrências PDBR dentro do banco de dados, que também pode ser representada, no todo ou em parte, por meio de indicadores de localização. Os indicadores de localização envolvidos são os indicadores de localização gêmeos no prefixo da raiz. As possibilidades são.

- Em HDAM, todas as raízes que colidem em uma posição K (veja a seção 19.5) são mantidas em uma cadeia (de um ou dois sentidos) que se inicia no registro K e é mantida em seqüência ascendente de raiz. (Isto é somente uma representação parcial de ocorrências PDBR, como veremos na seção 19.5.)
- Em HIDAM, se for usada a opção de um só sentido, todas as raízes dentro de um registro armazenado são encadeadas, mas em seqüência cronológica inversa, não na seqüência própria das raízes; isto é, a raiz mais recentemente inserida estará no início da cadeia. Neste caso, é o índice do HIDAM que define a seqüência de ocorrências PDBR (veja a Seção 19.6); a cadeia serve para uso próprio do IMS, e não permite qualquer acesso direto do usuário.
- Em HIDAM, se for usada a opção de dois sentidos, todas as raízes no banco de dados são mantidas na seqüência própria das raízes na cadeia de dois sentidos, permitindo assim recuperação seqüencial das raízes sem referência ao índice (veja a Seção 19.6).

Cada uma das cadeias aqui mencionadas usa indicadores de localização gêmeos do segmento raiz. Isto é verdade independentemente de serem especificados indicadores de localização hierárquicos ou filhos/gêmeos (na porção de mapeamento do DBD; veja a Seção 19.8).

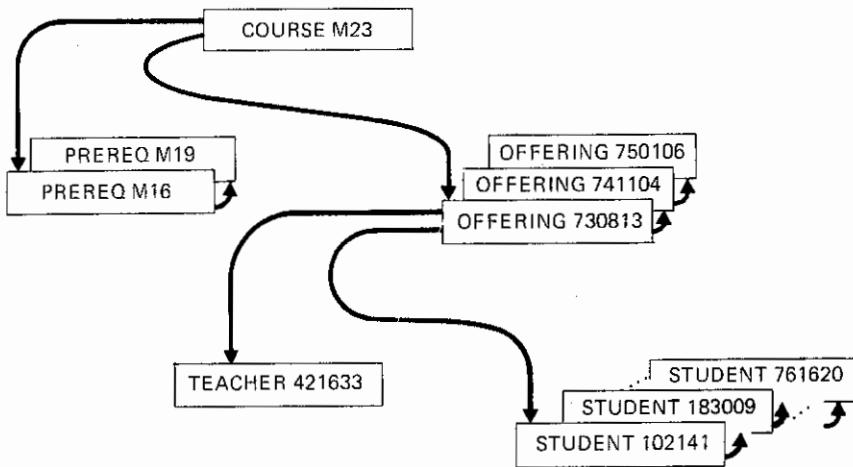


Fig. 19.8 Exemplo de indicadores de localização filho/gêmeo.

Finalmente, observemos que é possível misturar os dois tipos de indicadores de localização dentro de um tipo PDBR; isto é, podem ser especificados indicadores de localização hierárquicos para alguns segmentos na hierarquia, e indicadores de localização filhos/gêmeos para outros. A descrição detalhada das vantagens e desvantagens de desempenho relativo que possam influenciar na escolha de um arranjo específico de indicadores de localização está além do escopo deste livro. No entanto, como o acesso aos segmentos subordinados tem que ser obviamente seqüencial com indicadores de localização hierárquicos, enquanto é (relativamente) direto com indicadores de localização filhos/gêmeos, podemos dizer que, como regra geral, os indicadores de localização hierárquicos devem ser escolhidos se a maior parte do processamento for de natureza seqüencial, e os filhos/gêmeos em caso contrário. Observe, entretanto, que os indicadores de localização filhos/gêmeos normalmente ocuparão mais espaço do que os indicadores de localização hierárquicos.

19.5 HDAM

O HDAM fornece acesso direto (pelo valor do campo de ordenação) aos segmentos raiz, via técnicas de randomização ou encadeamento, juntamente com acesso por indicadores de localização aos segmentos subordinados (como já vimos). Em sua forma mais simples, um banco de dados HDAM consiste de um único arquivo OSAM ou um único arquivo VSAM por seqüência de entrada, divididos em registros armazenados de comprimento fixo. Os registros armazenados são numerados a partir de 1; os registros de 1 a N formam a “área endereçável do segmento raiz”, e os registros restantes formam uma área de excedentes. (O valor de N é especificado no DBD; veja a Seção 19.8.) A Fig. 19.9 mostra a estrutura de um banco de dados HDAM.

O HDAM difere das outras três estruturas por não ser necessário que a carga inicial do banco de dados seja feita em seqüência. Mais precisamente, os segmentos *raiz* podem ser apresentados para carga em qualquer ordem; no entanto, todos os segmentos depen-

dentes de uma determinada raiz têm que ser apresentados em estrita seqüência hierárquica antes da carga da próxima raiz.

Quando um segmento raiz é apresentado para carga, o valor do seu campo de ordenação é passado à rotina de randomização fornecida pelo DBA, que randomiza o valor e gera o endereço, K , de um registro dentro da área endereçável do segmento raiz (portanto $1 \leq K \leq N$). A nova raiz será colocada no registro K , desde que o registro K tenha espaço suficiente para ela. Se não, a nova raiz será colocada no registro mais próximo da área endereçável do segmento raiz que tenha espaço suficiente. Se não existir espaço em nenhum lugar da área endereçável do segmento raiz, a nova raiz será colocada na próxima posição disponível da área de excedentes. (Esta descrição está um pouco simplificada; veja a referência [19.1] para maiores detalhes.)

O procedimento descrito acima é seguido exatamente da mesma forma quando são apresentadas novas raízes para inserção em um banco de dados existente.

Os segmentos cujos valores de campo de ordenação randomizem para o mesmo valor K colidem em K (não é uma terminologia IMS). Todas essas colisões em K são mantidas em seqüência ascendente de raiz em uma cadeia que se inicia em um “ponto de ancoragem” dentro do registro K (na verdade, esta é a cadeia gêmea; veja a Seção 19.4). Observe que, em geral, algumas dessas colisões em K ficarão no registro K , algumas em outros registros dentro da área endereçável do segmento raiz, e algumas nos registros excedentes. Observe também que cada uma das N cadeias de colisões fica inteiramente separada de todo o restante — elas não são ligadas a nenhuma outra. (Por isso, em HDAM — dependendo da rotina de randomização fornecida pelo DBA — a seqüência lógica das ocorrências de segmentos raiz pode não estar completamente representada, pois duas raízes que sejam consecutivas em seqüência hierárquica provavelmente estarão em cadeias de colisão diferentes. No entanto, abaixo de qualquer raiz, a seqüência hierárquica das ocorrências de segmentos subordinados sempre estará completamente representada por meio dos indicadores de localização apropriados.)

Como exemplo, suponhamos que $N = 100$ e que o algoritmo de randomização seja “divida o valor do campo de ordenação da raiz por N , e tome $K = \text{resto mais } 1$ ”. Suponhamos que ocorra a seguinte seqüência de eventos.

- A raiz 322 é apresentada para inserção.
 $K = 23$; o registro 23 tem espaço disponível.

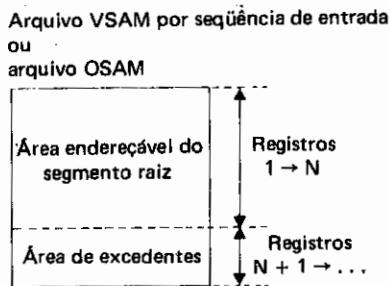


Fig. 19.9 Estrutura de um banco de dados HDAM.

- A raiz 222 é apresentada para inserção.
 $K = 23$; o registro 23 tem espaço disponível.
- A raiz 222 é apresentada para inserção.
 $K = 23$; o registro 23 está cheio; o registro mais próximo com espaço é o registro 25.
- A raiz 422 é apresentada para inserção.
 $K = 23$; a área endereçável do segmento raiz está cheia; a próxima posição disponível está na área de excedentes, no registro 144.

A cadeia de colisões aparecerá como mostrado na Fig. 19.10.

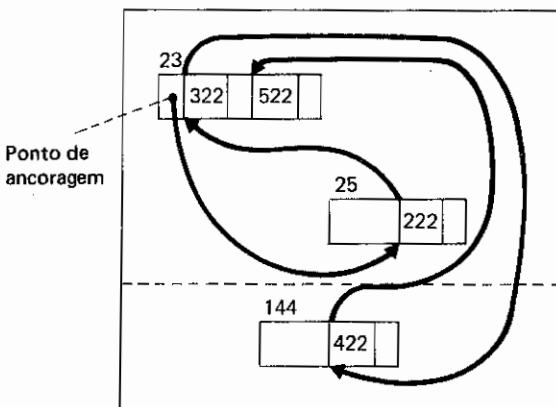


Fig. 19.10 Exemplo de uma cadeia de colisões (HDAM).

Podemos ver que o acesso aos segmentos raiz em HDAM será muito rápido, desde que as cadeias de colisão não se tornem muito longas. Sem dúvida, este é um dos objetivos principais da estrutura HDAM. É feita também uma tentativa para fornecer acesso rápido aos segmentos subordinados, colocando-os fisicamente próximos à raiz correspondente (reduzindo consequentemente o tempo de busca inerente aos dispositivos de cabeçote móvel). Suponhamos que a raiz resida no registro K . Quando é inserido o primeiro dependente (seja durante a carga ou mais tarde), ele será colocado o mais próximo possível da raiz, possivelmente no próprio segmento K ; O método usado para pesquisa de espaço é o mesmo que para a raiz que randomizou para K . Se a próxima operação DL/I for a inserção de outro dependente desta raiz, este dependente será também colocado o mais próximo possível daquela raiz. O processo continua até que (a) a série de inserções sob a raiz corrente termine – isto é, seja executada outra operação DL/I, talvez uma inserção envolvendo outra ocorrência PDBR; ou (b) a colocação de qualquer outro dependente na área endereçável do segmento raiz significasse que mais de M bytes desta área tivessem sido designados a esta ocorrência PDBR durante esta série de inserções, caso em que todas as inserções adicionais de dependentes desta série seriam colocadas na área de excedentes.

(O valor M , que é especificado no DBD, age como um limite de segurança para evitar que uma ocorrência PDBR excepcionalmente grande ocupe toda a área endereçável do segmento raiz durante a carga.) Todo o processo é repetido na próxima vez em que for executada uma série de inserções envolvendo esta ocorrência PDBR.

Observe que a inserção de um novo segmento não provoca movimentação de qualquer segmento já existente em HDAM — uma vantagem significativa de desempenho em relação ao HISAM. Naturalmente isto é possível graças aos indicadores de localização, permitindo que as rotinas de alocação de espaço coloquem um segmento inserido em qualquer posição conveniente e ainda assim mantendo tanto a seqüência hierárquica quanto a possibilidade de acesso direto ao segmento. A remoção de segmentos também é feita de forma diferente nas duas estruturas. Em HISAM, as remoções são simplesmente registradas pela colocação de um indicador no prefixo do segmento. Em HDAM, entretanto, o espaço ocupado por um segmento removido torna-se disponível para ser novamente usado, e assim uma inserção subsequente pode ser fisicamente gravada sobre o removido.

19.6 HIDAM

HIDAM fornece acesso indexado aos segmentos raiz, e acesso por indicadores de localização aos segmentos subordinados. Tal como em HISAM, a indexação é no campo de ordenação do segmento raiz. Em HIDAM, entretanto, o índice é controlado pelo IMS, e não pelo método de acesso (veja observações sobre indexação controlada pelo DBMS no Capítulo 2). Um banco de dados HIDAM consiste de *dois* bancos de dados: um banco de dados “dados”, que contém realmente os dados, e um banco de dados ÍNDICE, que fornece os índices (densos).

O banco de dados “dados” consiste, na sua forma mais simples, de um único arquivo OSAM ou um único arquivo VSAM por seqüência de entrada, dividido em registros armazenados de comprimento fixo. A carga inicial do banco de dados tem que ser feita em seqüência hierárquica. Cada segmento apresentado é colocado em seqüência na próxima posição disponível, como se este fosse um banco de dados HSAM. Naturalmente, além disso, os valores apropriados de indicadores de localização são colocados em prefixes relevantes. As remoções subsequentes são manuseadas exatamente como em HDAM; isto é, o espaço é liberado e pode ser usado para segmentos que venham a ser adicionados posteriormente. A inserção subsequente de um segmento raiz faz com que a nova raiz seja colocada o mais próximo possível da raiz que a precede na seqüência hierárquica. A inserção subsequente de um segmento dependente faz com que o novo segmento seja colocado o mais próximo possível de seu predecessor imediato na seqüência hierárquica. (Como em HDAM, uma vez armazenados os segmentos nunca se movem.) Para maiores detalhes sobre o método usado para pesquisa de espaço veja a referência [19.1].

O banco de dados ÍNDICE é uma forma especial de banco de dados HISAM — isto é, ou um par de arquivos ISAM/OSAM ou um único arquivo VSAM por seqüência de chaves (não é necessário arquivo por seqüência de entrada). Ele contém apenas um tipo de segmento, o segmento índice.¹⁰ Há uma ocorrência de segmento índice para cada ocorrência de raiz do banco de dados “dados”; esta contém o valor do campo de ordenação da raiz, juntamente com um indicador da localização da raiz. Este indicador de localiza-

¹⁰ O banco de dados ÍNDICE é portanto “somente de segmentos raiz”, razão por que não é necessário arquivo por seqüência de entrada no caso VSAM.

ção faz parte do prefixo do segmento índice (é, de fato, um indicador de localização filho; veja o Capítulo 20).

A Fig. 19.11 mostra como o exemplo da Fig. 19.5 (a) apareceria em HIDAM, usando HISAM/OSAM. Se fosse usado VSAM, o banco de dados “dados” consistiria de um arquivo VSAM por seqüência de entrada; o banco de dados ÍNDICE consistiria de um arquivo VSAM por seqüência de chave (somente), e todos os segmentos índice estariam contidos nele, incluindo o de M26. (Não existiria o indicador deste segmento no segmento índice de M27.)

19.7 GRUPOS DE ARQUIVOS SECUNDÁRIOS

Até agora consideramos que cada banco de dados armazenado consistia de um único “grupo de arquivos” (DSG). No entanto, os bancos de dados HISAM, HDAM, ou HIDAM – não os bancos de dados HSAM ou ÍNDICE – podem ser divididos em um grupo de arquivos *primário* e de um a nove grupos de arquivos *secundários*. Cada DSG consiste de um par de arquivos ISAM-OSAM (para HISAM),¹¹ ou de um único arquivo OSAM ou VSAM por seqüência de entrada (para HDAM e HIDAM). Em cada caso, todas as ocorrências de qualquer tipo de segmento estão contidas totalmente dentro de um DSG (por outro lado, diversos tipos de segmentos diferentes podem estar contidos no mesmo DSG). O DSG primário é o que contém as raízes.

Em HISAM, a vantagem básica (não a única) de se dividir um banco de dados em vários DSGs é que cada DSG secundário fornece acesso indexado, via o índice ISAM naquele DSG, a alguns segmentos dependentes. Isto traz como efeito a melhora do tempo de resposta de algumas operações de acesso direto (por exemplo, um “get unique” para um segmento próximo ao canto direito do fundo da hierarquia não precisa envolver a travessia de todos os segmentos que o antecedem na hierarquia). Outras vantagens podem advir do fato de poder o banco de dados se distribuir por vários dispositivos, permitindo (por exemplo) a concentração de porções do banco de dados com grande atividade em áreas comparativamente pequenas, e a possibilidade de maior concorrência de acesso a diferentes porções. Por outro lado, o tempo de resposta de algumas operações pode sofrer aumento, devido (por exemplo) ao espaço adicional requerido para os índices extra.

A Fig. 19.12 mostra como o banco de dados de educação da Fig. 16.1 poderia ser dividido em dois DSGs, o primário contendo os segmentos COURSE e PREREQ, e o (único) secundário contendo os segmentos OFFERING, TEACHER e STUDENT. A Fig. 19.13 mostra como parte deste banco de dados poderia parecer logo após a carga (conforme Fig. 19.4).

Sob HISAM, a divisão em múltiplos DSGs é geralmente executada como a seguir.

1. Escolha um tipo de segmento de segundo nível (isto é, um filho imediato da raiz.)
2. Designe aquele segmento e todos os que o seguem na estrutura hierárquica para um DSG secundário.
3. Remova esses segmentos da hierarquia e repita as etapas 1–3 sobre a hierarquia remanescente (se desejado).

¹¹

Não podem ser usados DSGs secundários em HISAM se o método de acesso de suporte for VSAM.

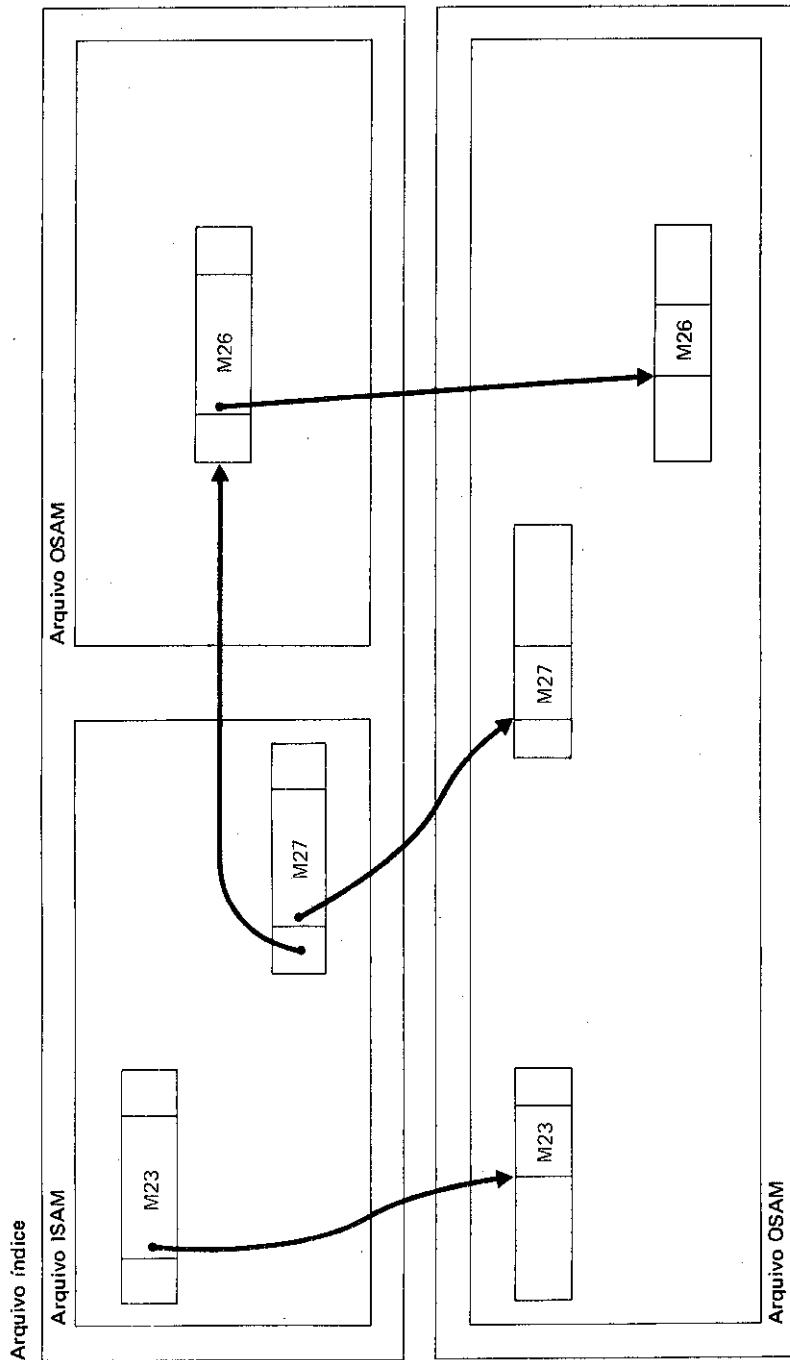


Fig. 19.11 Exemplo de inserção de segmento raíz (HISAM, usando HISAM/OSAM).

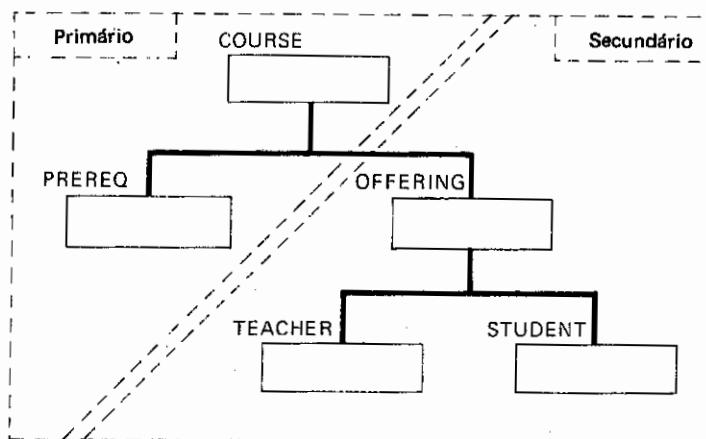


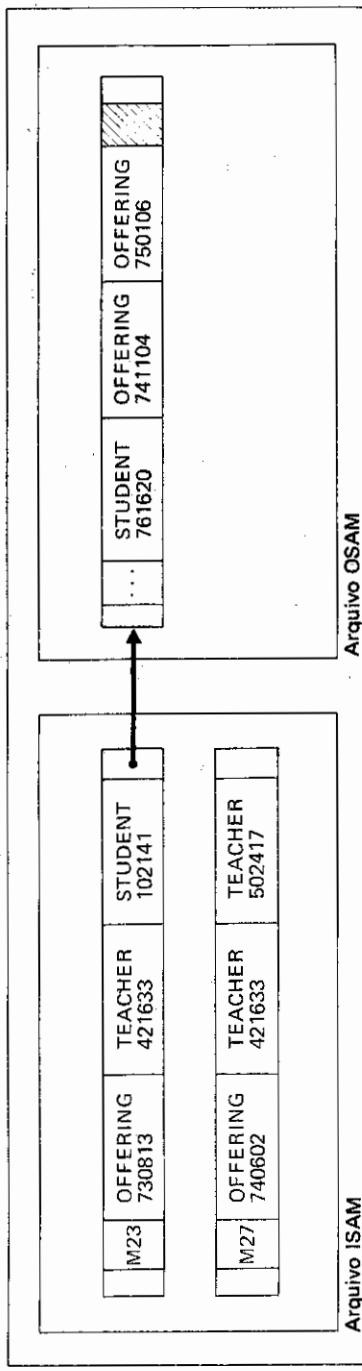
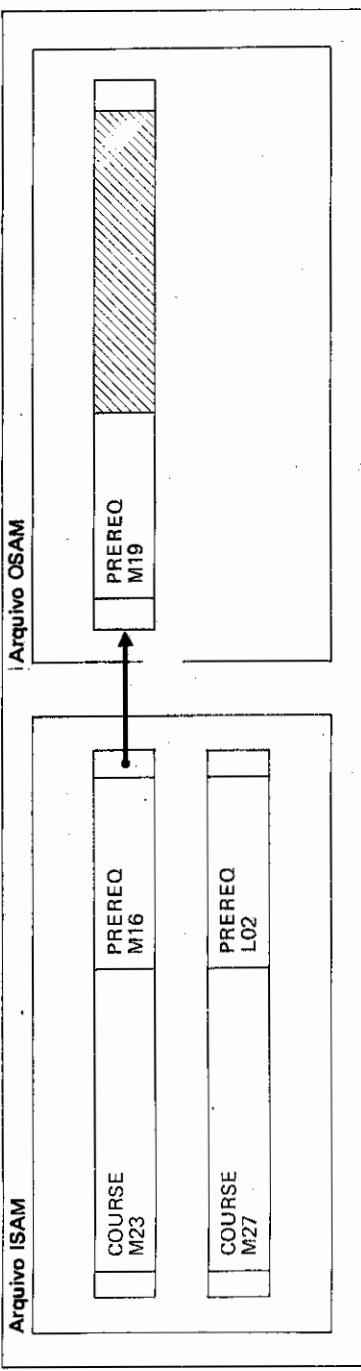
Fig. 19.12 Divisão do banco de dados de educação em dois DSGs (HISAM).

Observe que cada DSG secundário começa com um segmento de segundo nível. O que ocorre quando o banco de dados é carregado é o seguinte: os segmentos do DSG primário são carregados (COURSES e PREREQs no exemplo) exatamente como se fossem os únicos segmentos em uma situação de DSG único; os segmentos do DSG secundário (OFFERINGS, TEACHERs, e STUDENTs no exemplo) são carregados em seu DSG exatamente da mesma maneira, com a exceção de que os segmentos pertencentes a uma mesma ocorrência PDBR são precedidos em seu DSG por uma cópia do campo de ordenação da ocorrência raiz correspondente. Isto efetivamente atua como uma “raiz” do DSG secundário (veja Fig. 19.13). O índice ISAM no DSG secundário é um índice nesta “raiz”.

Para HDAM e HIDAM existe uma outra razão para se dividir o banco de dados em diversos DSGs (além das vantagens já mencionadas que derivam da distribuição dos dados por diversos dispositivos): pode ajudar a reduzir a fragmentação de memória. Como um exemplo algo simplificado, suponhamos que um banco de dados HD contenha dois tipos de segmentos, A (100 bytes) e B (60 bytes), e que o banco de dados consista de um único DSG. Suponhamos que é removido um segmento A, tornando disponível um espaço de 100 bytes para futuras inserções. Suponhamos agora que um segmento B é inserido e armazenado nesse espaço. Esta inserção deixará 40 bytes que não poderão ser usados, ou seja – 40 bytes totalmente perdidos. Se, por outro lado, os segmentos A e B forem designados para DSGs diferentes, este problema não poderá surgir; somente um segmento A poderá usar o espaço liberado pela remoção de um segmento A, e os segmentos B serão sempre inseridos em outro DSG.

Em HDAM/HIDAM não há restrições quanto à designação de tipos de segmentos a DSGs (em contraste com a situação de HISAM). Por exemplo, se o banco de dados de educação fosse armazenado em HDAM ou HIDAM, COURSES poderiam ser designados a um DSG (o primário), OFFERINGS a outro (um DSG secundário), e PREREQs, TEACHERs, e STUDENTs a um terceiro (outro DSG secundário). No caso de HDAM,

DSG primário



OSG secundário

Fig. 19.13 Parte do banco de dados de educação (HISAM, dois DSGs).

somente o DSG primário contém uma área endereçável de segmento raiz; todos os outros consistem inteiramente de área de excedentes. Existe uma restrição, de que segmentos em DSGs diferentes não podem ser conectados por indicadores de localização hierárquicos; neste caso *têm* que ser usados indicadores de localização filho/gêmeo.

As inserções e remoções em um DSG secundário são manuseadas essencialmente da mesma forma que em um DSG primário. Para maiores detalhes veja [19.1].

19.8 A DEFINIÇÃO DE MAPEAMENTO

Como mencionamos na Seção 16.2, o mapeamento de memória de um PDB é definido como parte do DBD. Podemos agora analisar este assunto com maiores detalhes. A definição de mapeamento envolve entradas adicionais nas instruções DBD e SEGM, e duas instruções adicionais (LCHILD e DATASET).

As entradas adicionais na instrução DBD são:

- **ACCESS = HSAM ou HISAM ou HDAM ou HIDAM ou INDEX**

Esta entrada especifica a estrutura relevante. Observe que INDEX (índice) é uma das possibilidades; para HIDAM são exigidos *dois* DBDs, um para o banco de dados “dados” (para o qual tem que ser especificado ACCESS = HIDAM) e um para o banco de dados ÍNDICE. Esses dois DBDs são referenciados como HIDAM DBD e INDEX DBD, respectivamente. Esses dois DBDs se referenciam um ao outro. (No entanto, observe que o usuário não tem que tomar conhecimento do banco de dados INDEX; o INDEX DBD não faz parte do “esquema conceitual” — faz parte somente da definição de mapeamento.)

Outros parâmetros na entrada ACCESS, não mostrados acima, especificam o método de acesso básico (por exemplo, OSAM ou VSAM). Pode ser também especificada uma variação do HSAM, conhecida como “HSAM simples” (SHSAM); um banco de dados HSAM simples é um banco de dados HSAM contendo somente um tipo de segmento, e não tem quaisquer prefixos armazenados. O HISAM tem uma forma “simples” análoga, conhecida como SHISAM (só disponível com VSAM, não com ISAM/OSAM).

- **RMNAME =**

Esta entrada só é exigida se ACCESS = HDAM. Ela especifica o nome da rotina de randomização fornecida pelo DBA (que tem que existir em uma biblioteca do sistema disponível ao IMS). Especifica também os valores N e M que são, respectivamente, a quantidade de registros armazenados na área endereçável de segmento raiz e a quantidade máxima de bytes da área endereçável de segmento raiz que pode ser designada a uma ocorrência PDBR durante uma série de inserções (veja a Seção 19.5).

As entradas adicionais na instrução SEGM relacionam-se com as opções de indicadores de localização disponíveis em HDAM e HIDAM; elas não se aplicam aos bancos de dados HSAM, HISAM, ou INDEX. Das entradas adicionais, a primeira e mais importante é o operando **POINTER**.

- **POINTER = HIER ou HIERBWD ou TWIN ou TWINBWD**

HIER especifica que cada ocorrência do segmento deve conter um indicador da localização da próxima ocorrência do segmento (de qualquer tipo) em seqüência hierárquica, com exceção da última ocorrência de segmento dependente sob uma determinada raiz, que *não* contém um indicador da localização da próxima raiz. Além disso, se o segmento sendo definido for a raiz, cada ocorrência incluirá também um indicador da localização da

próxima¹² ocorrência raiz. HIERBWD é o mesmo que HIER, com exceção de serem os indicadores de localização bidirecionais. TWIN especifica que cada ocorrência deste segmento deve conter um indicador da localização da próxima ocorrência (se houver) do mesmo tipo de segmento sob a mesma ocorrência pai e um indicador da localização da primeira ocorrência de cada tipo de filho sob esta ocorrência. Se o segmento sendo definido for a raiz, o indicador de localização gêmeo de cada ocorrência irá indicar a localização da próxima¹³ ocorrência raiz. TWINBWD é o mesmo que TWIN, mas tendo indicadores de localização gêmeos bidirecionais.

A segunda nova entrada na instrução SEGM consiste de um novo parâmetro no operando PARENT. Só é exigida se estiverem sendo usados indicadores de localização filho/gêmeo (POINTER = TWIN/TWINBWD para o pai) e um indicador da localização da última (bem como da primeira) ocorrência deste tipo específico de filho, para cada ocorrência de pai, sob esta ocorrência de pai (se desejado). A entrada é especificada para o segmento filho específico envolvido, sendo escrito como se segue (os parêntesis duplos são exigidos):

- PARENT = ((pai, DBLE))

A instrução LCHILD é usada primariamente em conexão com bancos de dados lógicos (serão discutidos no Capítulo 20). Entretanto, ela também é usada em HIDAM para ligar o segmento índice no banco de dados INDEX ao segmento sendo indexado no banco de dados "dados" correspondente; o segmento de dados é considerado como um "filho lógico" do segmento INDEX. O INDEX DBD deve conter uma instrução SEGM e uma instrução FIELD. Por exemplo, parte do INDEX DBD para o banco de dados da Fig. 19.11 poderia ser:

```
SEGMENT NAME=XSEG, BYTES=3  
FIELD NAME=(COURSE#, SEQ), BYTES=3, START=1
```

(Incidentalmente, o nome do campo não tem que ser o mesmo de campo especificado no HIDAM DBD, embora o seja neste exemplo.) Além destas duas instruções, o INDEX DBD tem que incluir uma instrução LCHILD (antes ou depois da instrução FIELD) para especificar o segmento de dados e o campo dentro deste no qual deverá ser executada a indexação (o qual tem que ser o campo de ordenação). Por exemplo,

```
LCHILD NAME=(COURSE, EDUCPDBD), INDEX=COURSE#
```

Tem também que ser incluída uma instrução LCHILD no HIDAM DBD, antes ou depois das instruções FIELD da raiz HIDAM. No nosso exemplo poderíamos ter

```
LCHILD NAME=(XSEG, XDBD), POINTER=INDX
```

onde XDBD é o nome do INDEX DBD. A entrada "POINTER = INDX" é exigida.

A outra instrução adicional é a DATASET. As instruções DATASET são usadas com dois objetivos principais. Primeiro, elas especificam, pelo seu posicionamento em re-

¹² Veja a Seção 19.4 para detalhes sobre o que significa "próximo" neste contexto.

¹³ Veja a Seção 19.4 para detalhes sobre o que significa "próximo" neste contexto.

lação às instruções SEGM no DBD, que segmentos serão designados a que grupos de arquivos. Segundo, especificam os nomes das instruções do OS/VIS Job Control Language que serão necessárias quando uma aplicação for operar sobre o banco de dados. Vamos analisar cada uma dessas funções.

Para HISAM, tem que ser fornecida uma instrução DATASET para cada grupo de arquivos. Cada uma deve preceder imediatamente as instruções SEGM dos segmentos naquele DSG. A Fig. 19.14 mostra a seqüência de instruções exigida para definir o banco de dados da Fig. 19.12 (só estão mostradas as instruções e entradas relevantes).

```
DBD      NAME=EDUCPDBD, ACCESS=HISAM
DATASET ...
SEGM    NAME=COURSE, ...
SEGM    NAME=PREREQ, ...
DATASET ...
SEGM    NAME=OFFERING, ...
SEGM    NAME=TEACHER, ...
SEGM    NAME=STUDENT, ...
```

Fig. 19.14 Exemplo do posicionamento das instruções DATASET (HISAM).

Para HDAM e HIDAM, pode ser necessário mais de uma instrução DATASET para um determinado DSG, porque os segmentos não têm que estar adjacentes uns aos outros na estrutura hierárquica (que é definida pelas instruções SEGM) para serem colocados no mesmo DSG. Nessa situação, são usadas instruções DATASET rotuladas. Tal como em HISAM, cada segmento é designado ao DSG identificado pela instrução DATASET precedente mais próxima; entretanto, duas (ou mais) instruções DATASET com o mesmo rótulo são consideradas como se referenciando ao mesmo DSG. A Fig. 19.15 mostra as instruções necessárias para se definir uma estrutura HDAM para o banco de dados de educação consistindo de três DSGs, um contendo COURSES, um contendo OFFERINGS, e um contendo PREREQS, TEACHERS, e STUDENTS. PRIME, SECONDA, e SECONDB são rótulos.

```
DBD      NAME=EDUCPDBD, ACCESS=HDAM, ...
PRIME   DATASET ...
        SEGMENT NAME=COURSE, ...
SECONDA DATASET ...
        SEGMENT NAME=PREREQ, ...
SECONDB DATASET ...
        SEGMENT NAME=OFFERING, ...
SECONDA DATASET (blank)
        SEGMENT NAME=TEACHER, ...
        SEGMENT NAME=STUDENT, ...
```

Fig. 19.15 Exemplo de posicionamento das instruções DATASET (HD).

Uma instrução DATASET com o mesmo rótulo de uma instrução DATASET precedente não pode conter qualquer outra entrada.

A segunda função executada pela instrução DATASET é a de especificar os “ddnames” apropriados da Job Control Language. Na execução de uma aplicação sobre o banco de dados, o OS/VS exige “instruções DD”, identificadas por ddnames, para cada arquivo em cada DSG. Uma instrução DD define os detalhes finais do mapeamento do arquivo na memória física. Por exemplo, especifica o dispositivo no qual está montado o volume contendo o arquivo (que pode variar de corrida para corrida). Falando genericamente, esses detalhes formam (pelo menos em parte) a definição do mapeamento entre os interfaces dos registros físico e armazenado, e nós não iremos nos aprofundar mais sobre eles aqui. O leitor que estiver interessado deve pesquisar [19.3]. Vamos nos satisfazer observando que, quando vai ser rodada uma aplicação IMS, o DBA ou o usuário tem que fornecer todas as instruções DD apropriadas, com os ddnames conforme especificados nas instruções DATASET. As entradas DATASET relevantes são:

- DD1 = ddname

Para HSAM, este é o ddname do arquivo de entrada (SAM) que forma um banco de dados existente (“mestre anterior”). Para um banco de dados HISAM DSG ou INDEX é o ddname do arquivo ISAM ou do arquivo VSAM por seqüência de chave. Para HDAM ou HIDAM DSG é o ddname do arquivo OSAM ou do arquivo VSAM por seqüência de entrada.

- DD2 = ddname

Para HSAM (único caso em que esta entrada é necessária), este é o ddname do arquivo de saída (SAM) que forma um novo banco de dados (“novo mestre”).

- OVFLW = ddname

Para bancos de dados HISAM DSG ou INDEX usando ISAM/OSAM (únicos casos em que esta entrada é usada), este é o ddname do arquivo OSAM.

A instrução DATASET contém algumas poucas entradas adicionais referentes a detalhes de comprimento de registro, fatores de bloco e outras. Veja a referência [19.1].

19.9 REORGANIZAÇÃO

Fica aparente das Seções 19.3–19.6 que, à medida que o tempo passa, e mais e mais inserções e remoções são feitas no banco de dados, a organização do banco de dados na memória tornar-se-á mais e mais desordenada. Por exemplo, em HISAM mais e mais espaço será tomado por segmentos indesejáveis (removidos). (Naturalmente este não é um problema específico do IMS; a maioria dos sistemas tem problemas semelhantes.) À medida que se degenera a organização do banco de dados armazenado, o mesmo ocorre com o desempenho global do sistema, tanto em termos de utilização de espaço quanto de tempo de resposta. Eventualmente será atingido um ponto em que se torna necessário reorganizar o banco de dados para melhorar o desempenho — particularmente a utilização de espaço — evitando uma degeneração a níveis inaceitáveis. A reorganização é o processo de descarga e recarga do banco de dados. Mais precisamente, envolve a recuperação de todos os segmentos (não removidos) do banco de dados existente e sua carga em um novo banco de dados. (Na prática, devido a limitações da quantidade de dispositivos disponíveis, é usual descarregar-se o banco de dados anterior para fita — provavelmente na forma de um banco de dados HSAM — e depois recarregá-lo a partir desta, superpondo o banco de

dados anterior pelo novo. Esta técnica tem como vantagem adicional o preparo de uma cópia de *back-up* do banco de dados).

Na prática usa-se normalmente um programa utilitário do IMS para fazer a reorganização. Entretanto, para fins de explicação, vamos supor que o usuário tenha que escrever esse programa. Em geral, tudo o que é necessário é uma série de operações “get next” não qualificadas sobre o banco de dados anterior e uma série correspondente de operações “insert” no novo banco de dados. Isto irá recuperar todos os elementos existentes em sequência hierárquica (não é muito verdadeiro para HDAM, no qual a sequência só será hierárquica dentro de cada ocorrência PDBR) e carregá-los na mesma sequência no novo banco de dados.

Para HISAM, este processo irá recuperar o espaço ocupado pelos segmentos removidos, pois as remoções não serão gravadas no novo banco de dados. Sob ISAM/OSAM, irá também movimentar segmentos raiz do arquivo OSAM (inserções) para suas posições corretas no arquivo ISAM do novo banco de dados.

Para HDAM, a reorganização normalmente terá o efeito de mover os segmentos raiz da área de excedentes para a área endereçável do segmento raiz. Irá também recuperar o espaço perdido por fragmentação.

Para HIDAM, a reorganização irá colocar todos os segmentos em uma sequência física que reflete a sequência hierárquica. Também será recuperado o espaço perdido por fragmentação.

Como regra geral, não é necessário reorganizar um banco de dados HDAM ou HIDAM tão freqüentemente quanto um banco de dados HISAM, devido às diferenças entre as técnicas empregadas nos dois casos para inserção e remoção.

→ discussões do banco de dados

19.10 INDEPENDÊNCIA DE DADOS

Na seção anterior, nós partimos do princípio tácito de que quando é feita a reorganização, o DBD do novo banco de dados permanece essencialmente o mesmo do anterior. Na prática, são possíveis muitas diferenças entre os dois, isto é, a estrutura de armazenamento do novo banco de dados pode ser consideravelmente diferente da do anterior. Em outras palavras, o IMS oferece um alto grau de independência de dados. Abaixo estão algumas das variações possíveis.

- Podem ser adicionados novos tipos de segmentos em certos pontos da hierarquia (veja a Seção 17.1).
- Podem ser adicionados novos campos a tipos de segmentos existentes.
- Pode ser modificado o tamanho do registro armazenado.
- Podem ser escolhidas opções diferentes de indicadores de localização em HDAM ou HIDAM.
- Pode ser mudada a divisão do banco de dados em grupos de arquivos.
- Para HDAM, podem ser mudados os valores de N (quantidade de registros armazenados na área endereçável do segmento raiz) e M (quantidade máxima de bytes na área endereçável do segmento raiz que pode ser designada para uma única ocorrência PDBR em uma série de inserções).
- Pode ser modificada a rotina da randomização para HDAM.
- Pode ser modificado o método básico de acesso (por exemplo, em HDAM, OSAM pode ser substituído por VSAM).

Além disso, é possível converter-se um banco de dados de uma das quatro estruturas para outra — digamos de HISAM para HIDAM — sob algumas restrições. As possibilidades estão resumidas na Fig. 19.16.

Para \ de	HSAM	HISAM	HDAM	HIDAM
HSAM		OK	Nota 3	OK
HISAM	Nota 1		Nota 3	OK
HDAM	Nota 1 Nota 2	Nota 2		Nota 2
HIDAM	Nota 1	OK	Nota 3	

Fig. 19.16 Transformações do banco de dados armazenado em IMS.

Notas

1. Estas transformações são pouco prováveis na prática, pois o HSAM não permitirá operações “delete”, “replace”, ou “insert” (exceto na carga inicial, quando “insert” é a única operação válida).
2. O novo banco de dados tem que ser carregado em seqüência hierárquica. Esta exigência pode trazer dificuldades se a seqüência “gêmea” do segmento raiz — isto é, a seqüência definida pelos valores ascendentes do campo de ordenação do segmento raiz — não estiver adequadamente representada no banco de dados (HDAM) anterior. Veja a Seção 19.5.
3. O novo banco de dados (HDAM) provavelmente não conterá a representação da seqüência “gêmea” do segmento raiz (veja a nota 2). Conseqüentemente, operações “get next” ao nível de raiz geralmente não funcionarão em termos dessa seqüência. (Detalhes de como elas funcionarão estão fora do escopo deste livro.)

19.11 RESUMO

A Fig. 19.17 representa uma tentativa de se resumir as Seções 19.2—19.6, que formaram a maior parte do presente Capítulo. Observe que são usados indicadores de localização em HISAM, bem como em HDAM e HIDAM, mas que em HISAM eles estão sendo usados para ligar registros armazenados, não segmentos. A Seção 19.7 apresentou explicações sobre uma extensão das estruturas básicas de armazenamento, no conceito de grupo de arquivos secundários (não aplicável a HSAM). A Seção 19.8 mostrou como é definida a estrutura de memória para um PDB (via DBD). As Seções 19.9 e 19.10 discutiram o conceito de reorganização e os tipos de modificações que podem ser feitas quando um banco de dados é reorganizado. Finalmente, as vantagens e desvantagens das várias estruturas possíveis foram mencionadas no decorrer do Capítulo.

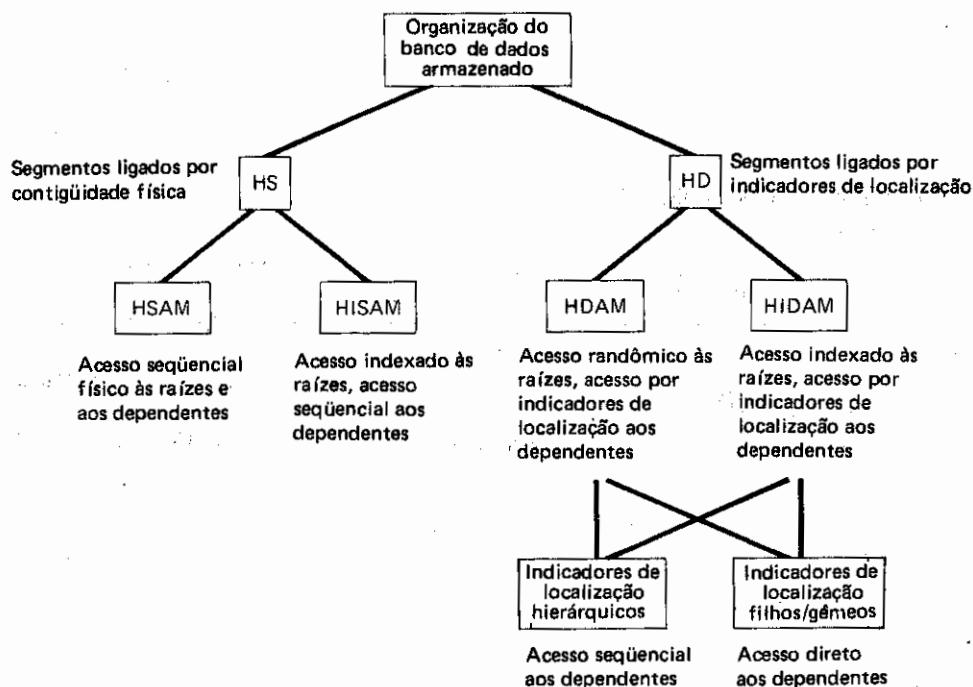


Fig. 19.17 As quatro estruturas básicas de armazenamento.

EXERCÍCIOS

19.1 Suponhamos que o banco de dados de educação deva ser armazenado em um banco de dados HDAM, com as opções de indicação de localização indicadas abaixo

```

DBD NAME=EDUCPDBD, ACCESS=HDAM, ...
SEGMENT NAME=COURSE, POINTER=TWIN, BYTES=256, ...
SEGMENT NAME=PREREQ, POINTER=HIER, BYTES=36, ...
SEGMENT NAME=OFFERING, POINTER=TWINBWD, BYTES=20, ...
SEGMENT NAME=TEACHER, POINTER=TWIN, BYTES=24, ...
SEGMENT NAME=STUDENT, POINTER=HIER, BYTES=25, ...
  
```

Desenhe um diagrama mostrando todos os indicadores de localização envolvidos com uma ocorrência PDBR da Fig. 16.2.

19.2 Cada prefixo em HDAM/HIDAM consiste de 1 byte de código de tipo, um byte de remoção, um contador de 4 bytes, e uma quantidade de indicadores de localização (4 bytes cada). Com base no DBD do Exercício 19.1, obtenha os valores relativos de espaço de memória necessário para prefixos e dados. Pode ser considerado que, em média, há dois PREREQs e oito OFFERINGS para cada curso, e 1,5 TEACHERs e 16 STUDENTS para cada OFFERING.

19.3 Novamente com base no DBD do Exercício 19.1 — seria possível dividir o banco de dados nos grupos de arquivos a seguir?

- COURSE, PREREQ (primário); OFFERING, TEACHER, STUDENT (secundário).
- COURSE, OFFERING, STUDENT (primário); PREREQ, TEACHER (secundário).

- c) COURSE, TEACHER (primário); PREREQ, OFFERING (secundário); STUDENT (outro secundário).

Nos casos em que for possível, mostre as instruções DATASET necessárias para efetuar a divisão. Quais dessas divisões seriam permissíveis se fosse usado HISAM ao invés de HDAM?

REFERÊNCIAS E BIBLIOGRAFIA

Veja também [16.1].

- 19.1 IBM Corporation. Information Management System/Virtual Storage System/Application Design Guide. IBM Form n° SH20-9025.
- 19.2 IBM Corporation. Operating System/Virtual Storage Access Method Services. IBM Form n° GC26-3836.
- 19.3 IBM Corporation. Operating System/Virtual Storage Job Control Language Reference Manual. IBM Form n° GC28-0618.
- 19.4 IBM Corporation. Operating System/Virtual Storage VSAM Programmer's Guide. IBM Form n° GC26-3838.

20

Bancos de Dados Lógicos do IMS

20.1 LOGICAL DATABASES (LDBs)

Já mencionamos diversas vezes que o termo “banco de dados lógico” tem dois significados diferentes em IMS. Com o primeiro significado nós lidamos no Capítulo 17. Este capítulo está voltado para o segundo, possivelmente o mais importante dos dois. De qualquer maneira é apenas uma introdução; maiores detalhes poderão ser encontrados nas referências [16.1] e [19.1].

Um banco de dados lógico então (LDB) – segundo significado – é um conjunto ordenado de ocorrências de registros lógicos do banco de dados (LDBR). Tal como um PDBR, um LDBR é um arranjo hierárquico de segmentos de comprimento fixo.¹ Tal como o PDB, um LDB é definido por meio de um DBD. Entretanto, o LDB difere do PDB por não ter existência própria; ao invés, é definido em termos de um ou mais PDBs existentes. Especificamente, cada ocorrência LDBR consiste de segmentos de diversas ocorrências PDBR distintas (de um ou vários PDBs). Assim, o LDB forma uma estrutura (hierárquica) dos dados que é diferente da estrutura (hierárquica) representada pelo(s) PDB(s) básico(s); em outras palavras, fornece ao usuário uma visão alternativa dos dados.

Podemos ver que os bancos de dados lógicos são, até certo ponto, o equivalente IMS das “visões” do Sistema R², e sem dúvida os objetivos globais das duas construções são semelhantes. Mas em IMS, o LDB é realmente parte do nível conceitual e não do nível externo. De fato, a estrutura de armazenamento representa diretamente o LDB, na medida em que contém indicadores de localização adicionais interligando os segmentos armazenados para formar a estrutura desejada (além dos indicadores de localização discutidos no capítulo anterior). Por isso, não é inteiramente verdadeiro dizer-se que o LDB propriamente não existe (mas o ponto é que os *dados* realmente pertencem a um ou mais

¹ Veja nota 1 de rodapé, Capítulo 16.

² Naturalmente o mesmo pode ser dito sobre o primeiro significado de bancos de dados lógicos.

PDBs, e não ao LDB). Sendo parte do nível conceitual, o LDB deve parecer um PDB ao usuário, e isto ocorre pelo menos no que tange às recuperações. A situação no que se refere a operações de atualização não é entretanto tão direta, por motivos que discutiremos nas Seções 20.5 e 20.6.

20.2 UM EXEMPLO

A seção de ornitologia de um instituto de conservação da natureza está conduzindo um levantamento sobre a vida dos pássaros em uma determinada região. A região foi dividida em áreas, e para cada área as observações deverão ser registradas em um banco de dados de levantamento. A informação registrada sobre uma área deve consistir de um número de área (identificador único), nome da área, descrição da área e, para cada tipo de pássaro observado na área, o nome comum, data da observação, notas do observador, nome científico do pássaro e um conjunto de informações descritivas. Assim, o registro do banco de dados, como visto pelo usuário, deve se parecer (aproximadamente) com o mostrado na Fig. 20.1.

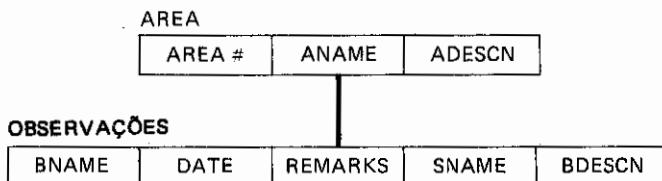


Fig. 20.1 Estrutura requerida para o banco de dados de levantamento.

O instituto já mantém, no entanto, um PDB com informações sobre pássaros. Este PDB é um banco de dados com “somente segmentos raiz”, tendo uma estrutura de registro como a mostrada na Fig. 20.2. Ali, BNAME (nome comum) é o campo de ordenação.

Dessa forma, se fosse montado um novo PDB com a estrutura mostrada na Fig. 20.1, a maior parte dos dados nele contidos seria uma repetição dos dados deste banco de dados de pássaros.

Além disso, o novo PDB teria um alto grau de redundância interna, já que muitas ocorrências PDBR conteriam informações idênticas – especificamente, o nome científico e as descrições seriam registrados uma vez para cada observação de um determinado tipo de pássaro, ao invés de somente uma vez.

BIRD		
BNAME	SNAME	BDESCN

Fig. 20.2 Estrutura do registro do banco de dados de pássaros.

Essas deficiências podem ser evitadas por meio de um banco de dados lógico, como a seguir. Primeiro, define-se um banco de dados *físico* com estrutura semelhante à mostrada na Fig. 20.1. No entanto, o segmento SIGHTING (observação) conterá apenas os campos DATE (data) e REMARKS (notas do observador), juntamente com um *indicador da localização* do segmento BIRD (pássaro) apropriado no PDB de pássaros. (Este indicador de localização faz parte do prefixo do segmento, e não será visto pelo usuário.) Teremos assim a situação ilustrada pela Fig. 20.3.

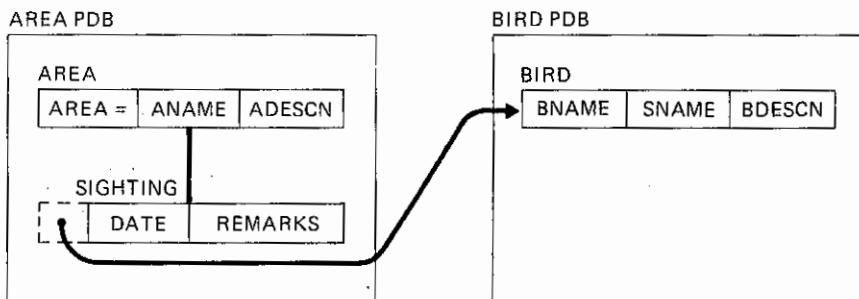


Fig. 20.3 PDBs AREA e BIRD.

Segundo, pode ser agora definido um banco de dados *lógico*, com a estrutura mostrada pela Fig. 20.4, que apresenta o que o usuário vê (compare com a Fig. 20.1). O usuário pode agora processar este LDB como se fosse um PDB — dentro de certos limites (a serem discutidos).

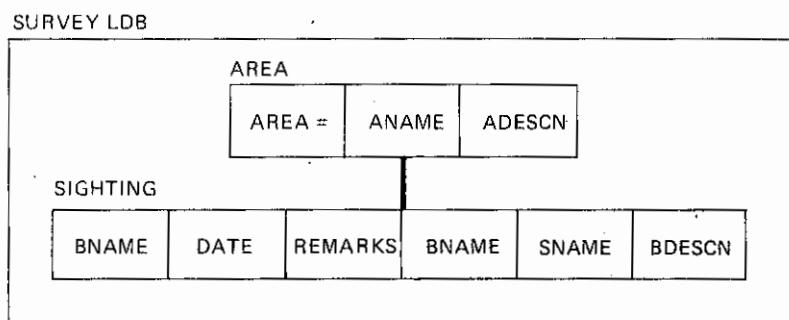


Fig. 20.4 O LDB SURVEY

Por exemplo, suponhamos que a Fig. 20.5 represente os dados existentes nos PDBs em determinado momento. Então, o LDB visto pelo usuário seria como o mostrado na Fig. 20.6. (Só estão mostrados os campos de ordenação, mas o leitor deve entender que cada ocorrência de segmento dependente inclui todos os campos de SIGHTING na Fig. 20.4. Observe que há *duas* observações para 'WREN' sob a área A3.)

A explicação anterior está algo simplificada. Para que um LDB possa ser definido em termos de um ou mais PDBs existentes, esses PDBs têm que estar apropriadamente definidos; isto é – seus DBDs têm que especificar que eles estão envolvidos com o LDB considerado. Em outras palavras, o DBA tem que estar ao par de que os PDBs serão envolvidos em um LDB no momento em que seus DBDs são escritos. Assim, no exemplo, ou foi antecipado que o PDB BIRD eventualmente participaria no LDB SURVEY, ou será necessário descarregar o PDB BIRD e depois recarregá-lo de acordo com o DBD revisado. O último costuma ser o método usado mais freqüentemente na prática. Veja a Seção 20.4.

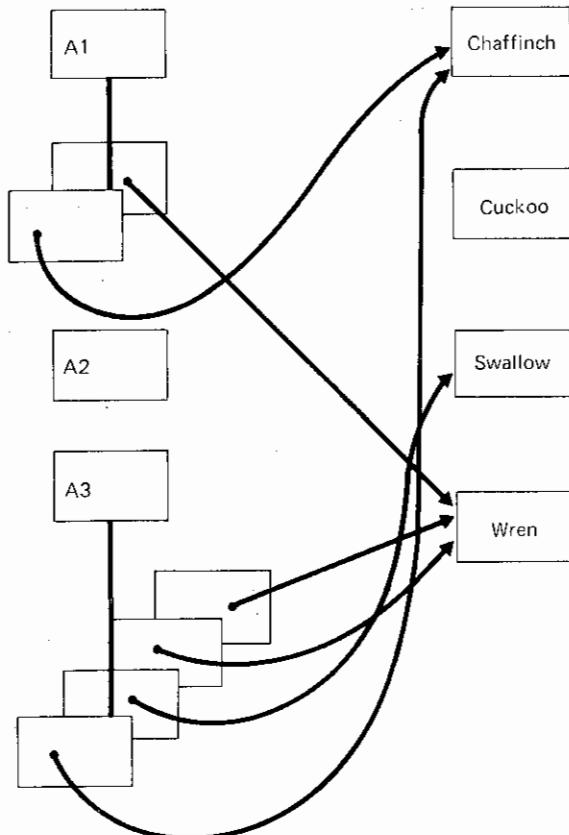


Fig. 20.5 PDBs de amostragem (AREA e BIRD).

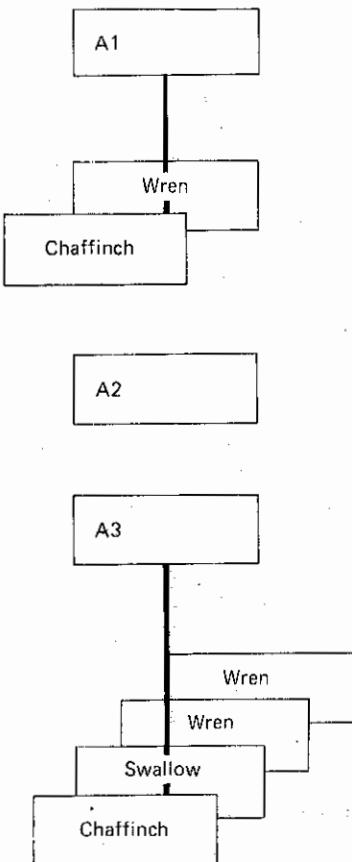


Fig. 20.6 LDB correspondente (SURVEY).

20.3 TERMINOLOGIA

O “segmento indicador de localização” SIGHTING da Fig. 20.3 é naturalmente um filho do segmento AREA. É também considerado como um filho do segmento BIRD (o segmento que ele indica). Portanto SIGHTING tem dois segmentos pais. Para fazer uma distinção entre os dois, nos referimos a AREA como o pai *físico*, e a BIRD como o pai *lógico*; também nos referimos a SIGHTING tanto como filho físico de AREA quanto como filho lógico de BIRD. O indicador de localização é um “pai lógico”. Um determinado segmento pode ter no máximo um pai físico e no máximo um pai lógico.

A terminologia de pai-lógico/filho-lógico, tal como a pai-físico/filho-físico, aplica-se tanto a tipos quanto a ocorrências. Assim, o segmento ‘CHAFFINCH’ de BIRD, por exemplo, é o pai lógico de dois segmentos SIGHTING, cujos pais físicos são, respectivamente, o segmento ‘A1’ de AREA e o segmento ‘A3’ de AREA. Além disso, esses dois segmentos SIGHTING são considerados como “gêmeos lógicos”; por analogia com gê-

meos físicos, todas as ocorrências de um tipo de filho lógico com um pai lógico comum são chamadas de gêmeos lógicos.

Como o exemplo da seção anterior ilustra, é permitido — embora não exigido — que o “segmento indicador de localização” contenha campos de dados em adição ao indicador de localização. Esse campo contém “dados de interseção” — isto é, dados que descrevem a combinação pai-físico/pai-lógico. Os dados de interseção devem normalmente ser funcionalmente dependentes da combinação das chaves totalmente concatenadas do pai físico e do pai lógico (veja o Capítulo 14).

Agora estamos prontos para uma explicação sobre a Fig. 20.4. O segmento SIGHTING no diagrama é uma concatenação dos três itens seguintes:

1. A chave totalmente concatenada do pai lógico
2. Os dados de interseção
3. O pai lógico (incluindo o valor do campo de ordenação do pai lógico)

Em geral, esta é a forma como um filho lógico sempre aparece para o usuário. (Na verdade é possível omitir-se *ou* os dois primeiros itens *ou* o terceiro, mas não faremos isso em nossos exemplos. O que *não* é possível é ver-se os dados de interseção sem a chave total concatenada do pai lógico.)

20.4 OS DATABASE DESCRIPTIONS (DBDs)

Um banco de dados lógico, tal como um banco de dados físico, é definido por meio de descrições do banco de dados (DBDs). Este DBD é chamado de “lógico”, em contraste com o DBD “físico”, que é o DBD de um banco de dados físico. Cada DBD lógico é definido em termos de um ou mais DBDs físicos básicos, que têm que existir. Assim, no nosso exemplo, são necessários (a) um DBD físico para o PDB BIRD, (b) um DBD físico para o PDB AREA, e (c) um DBD lógico para o LDB SURVEY. Vamos analisar cada um deles.

A Fig. 20.7 ilustra um esboço do DBD para o PDB BIRD.

```
DBD      NAME=BIRDPDBD, ...
SEGM    NAME=BIRD, POINTER=TWIN, ...
LCHILD  NAME=(SIGHTING, AREAPDBD)
        (FIELD statements for BIRD)
```

Fig. 20.7 DBD para o PDB BIRD (esboço).

O único a observar aqui é que a instrução SEGM do segmento BIRD está seguida por uma instrução LCHILD especificando o segmento SIGHTING — definido no DBD do PDB de AREA (Fig. 20.8) — como um filho lógico de BIRD. (Como lembrete, é útil mencionar que LCHILD aparece como parte da descrição do *pai* lógico. Em outras palavras, se a instrução SEGM de um tipo de segmento S for seguida por uma instrução LCHILD, isto significa que S *possui* um filho lógico, não que S é um filho lógico.)

A Fig. 20.8, que esboça um DBD possível para o PDB AREA, requer maiores explicações. Consideremos primeiro a entrada POINTER para o segmento SIGHTING, que especifica que o prefixo de SIGHTING deve conter os seguintes indicadores de localização:

- Um indicador da localização do pai lógico (conforme explicado na Seção 20.2)

```

DBD      NAME=AREAPDBD, ...
SEGMENT  NAME=AREA, POINTER=TWIN, ...
(FIELD statements for AREA)
SEGMENT  NAME=SIGHTING, POINTER=(LPARNT, TWIN),
          PARENT=((AREA), (BIRD, VIRTUAL, BIRDPDBD)), ...
FIELD    NAME=BNAME, ...
FIELD    NAME=DATE, ...
FIELD    NAME=REMARKS, ...

```

Fig. 20.8 DBD para o PDB AREA (esboço).

Isto naturalmente fornece a ligação básica necessária para a construção do LDB. Entretanto, só pode ser especificado LPARNT se o pai lógico residir em HDAM ou HIDAM; se residir em HISAM (HSAM não pode ser usado) terão que ser usados indicadores de localização simbólicos (veja abaixo).

- Um indicador de localização físico gêmeo (explicado no Capítulo 19).

Pode ser especificado TWIN (ou TWINBWD ou HIER ou HIERBWD), desde que o segmento SIGHTING resida em HDAM ou HIDAM; se este residir em HISAM, então naturalmente a seqüência física gêmea será representada por posição física. Mas qual é a seqüência física gêmea? Claramente gostaríamos que BNAME fosse o campo de ordenação para SIGHTING (veja a Fig. 20.6).⁴ Mas isto traz problemas adicionais, pois BNAME faz parte fisicamente do segmento BIRD, não do segmento SIGHTING. Voltaremos a este ponto daqui a pouco.

Consideremos a próxima entrada PARENT do segmento SIGHTING, que especifica (a) que o pai físico deste segmento é AREA, e (b) que o pai lógico deste segmento é BIRD, que está definido no DBD do PDB BIRD. VIRTUAL especifica que a chave totalmente concatenada do pai lógico é um campo virtual – campo BNAME – no que toca a este segmento, isto é, não está fisicamente armazenada como parte do segmento. Observe, entretanto, que *foi* fornecida uma instrução FIELD para BNAME.

A alternativa para VIRTUAL é PHYSICAL, significando que uma cópia do campo BNAME deve ficar fisicamente armazenada como parte do segmento SIGHTING. Se for especificado PHYSICAL, então (e só então) BNAME pode ser especificado (na instrução FIELD) como o campo de ordenação para este segmento. Assim, se for necessária ordenação, PHYSICAL *tem* que ser especificado na entrada PARENT. (A chave totalmente concatenada que será retornada no PCB na recuperação de um SIGHTING consistirá então de um valor AREA# seguido por um valor BNAME.)

Mas se for especificado PHYSICAL, o indicador de localização do pai lógico no prefixo fica conceitualmente redundante. Se o pai lógico residir em HDAM ou HIDAM, este indicador de localização ainda pode ser incluído por razões de desempenho, embora seja

4

Temos que especificar NAME = (BNAME, SQ.M) para permitir que haja múltiplas observações de um tipo de pássaro dentro de uma área (veja a Seção 16.2). Alternativamente, se (digamos) não tiver que ser registrada mais de uma observação por dia sobre cada tipo de pássaro dentro de cada área no banco de dados, a combinação de BNAME com DATE pode ser definida como campo de ordenação (e os valores seriam então únicos).

perfeitamente válido omiti-lo; entretanto, se o pai lógico residir em HISAM, o indicador de localização *tem* que ser omitido (e PHYSICAL tem que ser especificado). Nunca é possível o uso de indicadores de localização para indicar segmentos em HISAM, pois em HISAM a inserção de um novo segmento pode causar o movimento de segmentos existentes.

Portanto, em geral, temos duas maneiras distintas de representar a ligação de um filho lógico ao seu pai lógico (exceto quando o pai lógico reside em HISAM): ou via um indicador de localização "direto" (isto é, incluindo um indicador de localização do pai lógico no prefixo do filho lógico), ou via um indicador de localização "simbólico" (isto é, gravando fisicamente a chave totalmente concatenada do pai lógico como o primeiro registro do filho lógico). As vantagens relativas dessas duas maneiras estão discutidas em essência na Seção 2.4: Os indicadores de localização diretos permitem acesso mais rápido; os indicadores de localização simbólicos não precisam ser reajustados quando o banco de dados por eles indicados é reorganizado. A referência [19.1] discute outras considerações que afetam a escolha. É também possível usar-se uma combinação de ambas as técnicas. Entretanto, a escolha do método não afeta em nada o que o usuário vê no conteúdo do segmento lógico filho. Para maior simplicidade, consideraremos no restante deste Capítulo que todos os bancos de dados físicos estão em HDAM ou HIDAM, e que não serão usados indicadores de localização simbólicos (a menos de especificação explícita). Podemos agora examinar o DBD lógico (Fig. 20.9).

```
DBD      NAME=SVEYLDBD, ACCESS=LOGICAL
DATASET  LOGICAL
SEGM    NAME=AREA, SOURCE=((AREA,,AREAPDBD))
SEGM    NAME=SIGHTING, PARENT=AREA,
        SOURCE=((SIGHTING,,AREAPDBD),(BIRD,,BIRDPDBD))
```

Fig. 20.9 DBD para o LDB SURVEY.

Observe primeiramente que tem que ser especificado ACCESS = LOGICAL na instrução DBD, e que também tem que ser especificado LOGICAL na instrução DATASET. As instruções restantes definem os segmentos do LDB; note que não podem ser incluídas instruções FIELD. A primeira instrução SEGM estabelece que o segmento raiz do LDB, AREA, é de fato o segmento AREA definido no DBD do PDB AREA (o segmento poderia ter recebido um nome diferente dentro do LDB, se desejado). A vírgula dupla indica um operando omitido; veja detalhes na referência [16.1]. A segunda instrução SEGM estabelece que dentro deste LDB, SIGHTING é dependente de AREA, e consiste da concatenação do segmento SIGHTING definido no DBD do PDB AREA com o segmento BIRD definido no DBD do PDB BIRD. (Naturalmente, na visão do usuário, o segmento inclui como seu primeiro campo a chave totalmente concatenada de BIRD.)

20.5 CARGA DO BANCO DE DADOS LÓGICO

O processo de carga de um banco de dados lógico consiste essencialmente da carga do(s) banco(s) de dados físico(s) e colocação dos indicadores de localização requeridos. Por uma série de razões, esta operação é executada diretamente sobre o(s) PDB(s) básico(s),

e não no LDB propriamente dito. Para ver por que isto é necessário, suponhamos por um momento que o LDB SURVEY deva ser carregado via uma série apropriada de operações ISRT emitidas diretamente sobre aquele LDB. Surgiriam diversos problemas. Por exemplo, a inserção da ocorrência SIGHTING teria algumas vezes que provocar a criação de uma ocorrência BIRD, algumas vezes não, dependendo de já existir ou não a ocorrência BIRD no PDB BIRD. Também, o que ocorreria com um BIRD para o qual não haja SIGHTINGS (por exemplo, 'CUCKOO' na Fig. 20.5)? Como seria carregado? Como administrar a restrição de que os PDBs têm que ser carregados em sequência hierárquica (pelo menos em HISAM e HIDAM)? E novamente considerando os dados da Fig. 20.5, o que fazer se o valor BDESCN para 'CHAFFINCH' (digamos) estiver diferente nas SIGHTINGS A1 e A3?

Por razões como essas, o procedimento atualmente seguido é o de carregar-se o LDB inserindo segmentos diretamente no(s) PDB(s) básico(s), isto é, carregando cada um dos PDBs básicos como uma operação separada. Vamos supor que o PDB BIRD já foi carregado (na forma mostrada pela Fig. 20.5), e vamos examinar o que está envolvido na carga do PDB AREA.

AREA A1 é carregada (inserida) da forma usual. Pode então ser carregado um segmento SIGHTING, contendo os campos BNAME (com o valor 'CHAFFINCH'), DATE, e REMARKS, seguido por um segmento SIGHTING, semelhante para 'WREN'. Seguem-se depois AREA A2, AREA A3, uma SIGHTING para 'CHAFFINCH' sob A3, e SIGHTINGS semelhantes para 'SWALLOW' e 'WREN'. Observe, entretanto, que neste estágio os segmentos SIGHTING não contêm os indicadores de localização dos pais lógicos. Estes serão inseridos subsequentemente por um programa utilitário especial, que tem que ser executado antes que o processo de carga possa ser considerado como completo. Além disso, os prefixos dos pais lógicos têm que ser ajustados; isto também é manuseado por um programa utilitário especial, que opera sobre o PDB BIRD.

Para maiores detalhes sobre o processo de carga, veja a referência [16.1].

20.6 PROCESSAMENTO DO BANCO DE DADOS LÓGICO

Uma vez carregado o LDB, o usuário pode processá-lo exatamente como se este fosse um PDB, pelo menos no que concerne às operações de recuperação.⁵ No entanto, as operações de atualização são mais complicadas. Em geral, o efeito de uma operação de atualização sobre um segmento participante de um relacionamento lógico – isto é, um filho lógico, um pai lógico, ou um pai físico de um segmento que tenha também um pai lógico – é definido pela *regra* especificada pelo DBA para aquela operação e aquele segmento. O DBA tem que especificar, na instrução SEGM do DBD *físico* apropriado, uma regra de inserção, uma regra de remoção e uma regra de substituição para cada um desses segmentos. Cada uma dessas regras pode, em geral, ser definida como "física", "lógica", ou "virtual".⁶ Não pretendemos aqui entrar em maiores detalhes sobre essas regras (veja [16.1]), mas verificaremos sua significância em relação ao LDB SURVEY. Observe que as regras especificadas governam *todas* as operações de atualização do segmento; a operação em questão pode ser emitida (a) diretamente sobre o PDB relevante, ou (b) "indiretamente" sobre algum LDB construído sobre o PDB.

⁵ O LDB exige um PCB, como se fosse um PDB.

⁶ No caso de remoção há uma quarta possibilidade: "virtual bidirecional".

Consideremos primeiramente as operações ISRT. É sempre possível a inserção de uma AREA, e o efeito é simplesmente a inserção daquela AREA no PDB AREA. E sobre SIGHTINGS? Neste caso, como já mencionamos antes, o problema é que o BIRD ao qual se refere o SIGHTING pode ou não já existir no PDB BIRD. Se existir, o ISRT é aceito (isto é, o SIGHTING é inserido no PDB AREA); além disso, se a regra de inserção para BIRD for "virtual", os valores SNAME e BDESCN do segmento SIGHTING como apresentados pelo usuário substituem os valores anteriores do segmento BIRD correspondente. Se, por outro lado, o BIRD não existir correntemente no PDB de BIRD, o ISRT será (a) rejeitado se a regra de inserção para BIRD for "física"; (b) aceito nos outros casos, e então o efeito não será somente o de inserir o SIGHTING no PDB de AREA, mas também de inserir o BIRD no PDB de BIRD (novamente, lembre-se de que o SIGHTING apresentado pelo usuário inclui SNAME e BDESCN).

(Naturalmente é sempre possível a inserção de novos BIRDS no PDB de BIRD, independentemente do LDB SURVEY.)

Vamos agora considerar as operações de remoção. A remoção de uma AREA irá remover aquela AREA e seus subordinados SIGHTING do PDB de AREA. A remoção de uma SIGHTING (seja via remoção direta ou do seu pai AREA) removerá aquela SIGHTING do PDB de AREA, e se esta for a última SIGHTING do BIRD considerado, e BIRD tiver uma regra de remoção de "virtual", irá também remover o BIRD do PDB de BIRD.

Se for tentada uma remoção direta de BIRD do PDB de BIRD, o efeito dependerá da regra de remoção para BIRD. "Físico" só permitirá a remoção de BIRD se não houver SIGHTINGS se referenciando a ele. "Lógico" e "virtual" permitirão a remoção de um BIRD do PDB de BIRD, mantendo-o ao mesmo tempo disponível via o LDB SURVEY enquanto houver qualquer SIGHTING se referenciando a ele; (este efeito é conseguido por meio de indicadores apropriados no prefixo de BIRD). Este BIRD será inteiramente removido quando for removida a última SIGHTING que se refira a ele.

Finalmente, consideremos as operações REPL. A substituição de uma AREA causa simplesmente a correspondente substituição no PDB de AREA. A substituição de dados de interseção (DATE e REMARKS) em uma SIGHTING também causa simplesmente a ocorrência da substituição correspondente no PDB de AREA. A substituição de dados do pai lógico (SNAME e BDESCN) em uma SIGHTING só é permitida se a regra de substituição para BIRD for "virtual"; o efeito, neste caso, é a substituição do segmento BIRD, que por sua vez causa a mudança correspondente nos dados do pai lógico de todas as outras SIGHTINGS para aquele BIRD. (Naturalmente, um BIRD sempre pode ser substituído diretamente por meio de uma operação no PDB de BIRD.)

Portanto, resumindo, o LDB SURVEY pode se parecer mais ou menos com um PDB, desde que o DBA especifique regras como as mostradas na Fig. 20.10 (P, L, e V significam respectivamente "físico", "lógico", e "virtual").

Regra \ Segmento	AREA	SIGHTING	BIRD
Insert	P, L, ou V	V (Nota 1)	L ou V (Nota 3)
Delete	V (Nota 2)	P, L, ou V	P ou L (Nota 4)
Replace	P, L, ou V	P, L, ou V	V (Nota 3)

Fig. 20.10 Regras para os PDBs AREA e BIRD.

Notas

1. A regra de inserção de um segmento filho lógico é sempre dada como V.⁷
2. A regra de remoção de um segmento pai físico é sempre dada como V.
3. Observe que a estrutura do LDB SURVEY faz parecer que (por exemplo) para mudar o valor de BDESCN de um BIRD existente, seria necessário fazer a mudança em cada SIGHTING para aquele BIRD quando, de fato, isto não é exigido; isto é, a estrutura do LDB permite redundância na visão dos dados sem requerer uma redundância correspondente no que está armazenado.
4. A regra de remoção para BIRD poderia ser V, sem afetar o comportamento do LDB em si, mas isto permitiria que as operações no LDB (ou no PDB RAEA) trouxessem efeitos colaterais indesejáveis sobre o PDB BIRD. Por exemplo, a remoção de uma AREA poderia causar a remoção de um BIRD (se agora não houver SIGHTINGS para aquele BIRD).

Podemos ver que as regras só são críticas para o segmento BIRD (neste exemplo). Entretanto, tenha em mente que só discutimos o tipo mais simples possível de LDB, isto é, o que envolve um “relacionamento unidirecional”, com um pai físico, um pai lógico, e um filho lógico. O efeito das regras em situações mais complicadas foge ao escopo deste livro.

20.7 RELACIONAMENTOS LÓGICOS BIDIRECIONAIS

O LDB SURVEY, sendo hierárquico, representa a associação entre AREAs e BIRDs como se houvesse uma correspondência de um-para-muitos. De fato, naturalmente, a correspondência é realmente de muitos-para-muitos; não só há muitos BIRDs em uma determinada AREA, como também há muitas AREAs para um determinado BIRD. Assim (como foi mencionado no Capítulo 3), uma operação tal como “encontre todas as áreas nas quais foi observado um pássaro determinado” fica comparativamente difícil com o LDB SURVEY. Podemos superar esta dificuldade por meio de outro LDB — digamos BASURVEY — no qual AREA seja vista como uma subordinada de BIRD. (Por consistência, vamos mudar o nome do LDB original, no qual BIRD está subordinado a AREA, para ABSURVEY.) Assim teremos um “relacionamento lógico bidirecional” entre AREAs e BIRDs.

Os relacionamentos lógicos bidirecionais podem ser implementados no IMS de duas formas, respectivamente conhecidas como parelha física e parelha virtual.

Parelha Física

A parelha física envolve a introdução de um novo tipo de segmento — digamos BASIGHT — como filho físico de BIRD e filho lógico de AREA. (Novamente, por consistência, vamos mudar o nome original do segmento SIGHTING para ABSIGHT.) Veja a Fig. 20.11.

A finalidade do segmento BASIGHT é exatamente análoga à do segmento ABSIGHT, isto é: conter um indicador da localização (em seu prefixo) do seu pai lógico (AREA), juntamente com dados de interseção (DATE e REMARKS, como em

O IMS sempre assume V nesses casos, independentemente do que foi especificado, para evitar inconsistências que poderiam ocorrer de outra forma.

AREA PDB

BIRD PDB

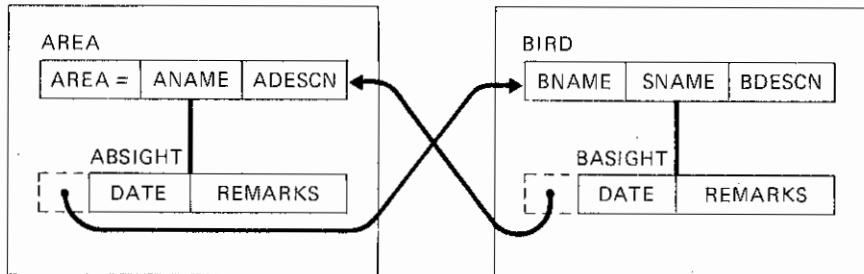


Fig. 20.11 PDBs de AREA e BIRD (com parelha física).

ABSLIGHT). Isto nos permite definir o LDB **BASURVEY** (bem como o LDB **ABSURVEY**). Uma ocorrência **BASIGHT**, como vista pelo usuário, consistirá da concatenação da chave totalmente concatenada do pai lógico (um valor **AREA#**), os dados de interseção, e os dados do pai lógico (incluindo novamente o valor **AREA#**). **ABSLIGHT** e **BASIGHT** são “segmentos parelha”, sendo declarados como tal ao IMS nos DBDs dos PDBs de AREA e BIRD; veja as Figs. 20.12 e 20.13.

A Fig. 20.12 mostra (nas linhas com asteriscos) as mudanças no DBD do PDB de AREA (compare com a Fig. 20.8). A instrução **SEGM** do segmento AREA está seguida por uma instrução **LCHILD** especificando o segmento **BASIGHT** – definido no DBD do PDB de BIRD (Fig. 20.13) – como um filho lógico de AREA.

```

DBD      NAME=AREAPDBD, ...
SEGMENT  NAME=AREA, POINTER=TWIN, ...
* LCHILD  NAME=(BASIGHT,BIRDPDBD), PAIR=ABSLIGHT
          (FIELD statements for AREA)
* SEGMENT  NAME=ABSLIGHT, POINTER=(LPARNT,TWIN,PAIRED),
          PARENT=((AREA),
          (BIRD,PHYSICAL,BIRDPDBD)), ...
FIELD    NAME=(BNAME,SEQ,M), ...
FIELD    NAME=DATE, ...
FIELD    NAME=REMARKS, ...

```

Fig. 20.12 DBD do PDB de AREA (esboço) – Parelha física.

A instrução **LCHILD** também especifica que o filho lógico (**BASIGHT**) forma uma parelha com o segmento **ABSLIGHT**, definido subseqüentemente no DBD corrente. Outra modificação neste DBD é a entrada **POINTER** de **ABSLIGHT**, que especifica **PAIRED** (parelha, entre outras coisas); esta especificação é exigida.

Observe, incidentalmente, que **BNAME** foi fisicamente incluído no segmento **ABSLIGHT**. Isto permite que as observações dentro de uma área sejam ordenadas por nome de pássaro.

A Fig. 20.13 mostra as modificações no DBD do PDB de BIRD (compare com a Fig. 20.7).

```

DBD      NAME=BIRD PDBD, ...
SEGMENT  NAME=BIRD, POINTER=TWIN, ...
* LCHILD NAME=(ABSIGHT, AREAPDBD), PAIR=BASIGHT
  (FIELD statements for BIRD)
* SEGMENT  NAME=BASIGHT, POINTER=(LPARNT, TWIN, PAIRED),
*           PARENT=((BIRD),
*                     (AREA, PHYSICAL, AREAPDBD)), ...
* FIELD    NAME=(AREA#, SEQ, M), ...
* FIELD    NAME=DATE, ...
* FIELD    NAME=REMARKS, ...

```

Fig. 20.13 DBD do PDB de BIRD (esboço) — Parelha física.

O DBD de BIRD é essencialmente semelhante ao DBD de AREA. Observe novamente que o campo do pai lógico na chave totalmente concatenada — AREA# — foi fisicamente incluído no segmento filho lógico (BASIGHT), para permitir que as observações sobre um pássaro sejam ordenadas por número de área.

O DBD do LDB ABSURVEY é o mesmo de antes (Fig. 20.9). O DBD do LDB BASURVEY é essencialmente semelhante.

Observe que a parelha física envolve armazenamento redundante dos dados de interseção, isto é: cada item dos dados de interseção será gravado fisicamente duas vezes, uma na ocorrência ABSIGHT e uma na ocorrência parelha BASIGHT. (Por exemplo, a ocorrência ABSIGHT que liga a área A1 a 'CHAFFINCH' em data específica contém precisamente os mesmos dados de interseção que a ocorrência BASIGHT que liga 'CHAFFINCH' à área A1 para aquela data.) No entanto, como os segmentos são parelhos, quando o usuário insere uma ocorrência em qualquer deles, o IMS automaticamente cria a ocorrência correspondente no outro (exceto durante a carga, quando é da responsabilidade do usuário criar ambos). De forma similar, se o usuário remove ou substitui uma ocorrência de um dos componentes da parelha, o IMS automaticamente faz a necessária modificação no outro.

Parelha Virtual

A parelha virtual evita a redundância da parelha física. A idéia é que pode ser eliminado um dos dois tipos de segmentos parelha; por exemplo, o segmento BASIGHT não precisa existir fisicamente, pois seus dados podem ser encontrados no segmento ABSIGHT. O relacionamento direcionado de BIRD para AREA fica então representado, não via um indicador de localização de pai lógico de BASIGHT para AREA, mas via um indicador de localização de filho lógico de BIRD para ABSIGHT *mais* um indicador de localização de pai físico⁸ de ABSIGHT para AREA. (São também requeridos indicadores de localização gêmeos lógicos; veja abaixo). A Fig. 20.14 ilustra esta organização.

⁸

Esta não é a única situação em que indicadores de localização do pai físico são requeridos. Também são necessários se um filho lógico indicar, via um indicador de localização direto, um pai lógico que não seja uma raiz. Nessa situação, cada segmento do caminho físico herárquico ao pai lógico incluirá um indicador de localização do pai físico, para permitir a montagem da chave totalmente concatenada do pai lógico.

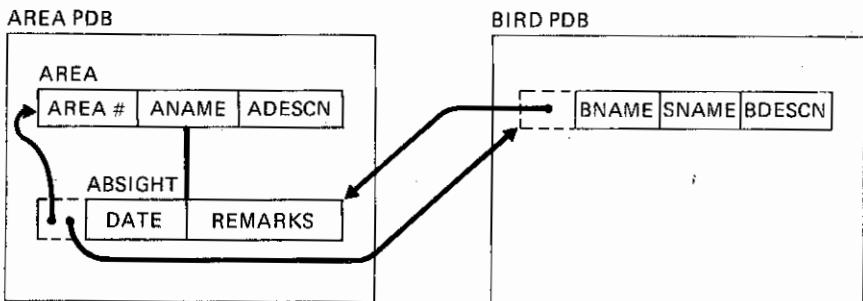


Fig. 20.14 PDBs de AREA e BIRD (com parelha virtual).

O primeiro ponto a ser observado é que a parelha virtual se parece quase exatamente com a parelha física para o usuário. No exemplo, embora não mais exista o segmento BASIGHT, o IMS faz com que pareça ainda existir; por esta razão, BASIGHT é dito ser um segmento virtual. (Há uma diferença de pouca importância entre um segmento virtual e um real, na visão do usuário: um segmento virtual não pode ser “carregado” como parte do processo de carga do PDB que o contém; ao invés disso, é “criado” quando o segmento real já se encontra carregado.) A representação convencional de uma situação de parelha virtual está mostrada na Fig. 20.15 (mas tenha em mente que a Fig. 20.14 representa o quadro real).

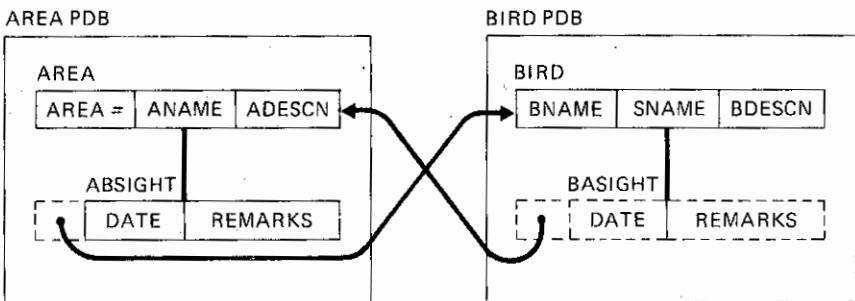


Fig. 20.15 PDBs de AREA e BIRD (com parelha virtual) — representação convencional.

Um segundo ponto geral é que o membro real da parelha (ABSIGHT no exemplo) tem que residir em HDAM ou HIDAM, pois o prefixo do segmento pai lógico contém um indicador direto da localização daquele, ou seja, um indicador da localização do filho lógico. (Os indicadores de localização dos pais lógicos são os únicos que podem ser simbólicos.)

Consideremos agora a ordenação existente neste exemplo. As observações dentro de uma área estão ordenadas por nome de pássaro; isto é feito pela inclusão física do campo BNAME no segmento ABSIGHT e a especificação de **POINTER = TWIN** (que liga todas

as ABSIGHTs de uma AREA em uma cadeia gêmea). E a ordenação de observações sobre um pássaro? Isto é representado por outra cadeia, a cadeia lógica gêmea, que liga todas as ABSIGHTs que indicam a localização do mesmo BIRD na seqüência da chave totalmente concatenada do pai físico. Dessa forma as observações sobre um pássaro ficam ordenadas por número de área, como requerido.

Conceitualmente, no entanto, esta segunda ordenação é obtida tornando AREA# o campo SEQ do segmento (virtual) BASIGHT, e é isto o especificado no DBD. Veja as Figs. 20.16 e 20.17.

```

DBD      NAME=AREAPDBD, ...
SEGMENT  NAME=AREA, POINTER=TWIN, ...
*        (* Não pode ser incluída instrução LCHILD para BASIGHT)
*        (* Instruções FIELD para AREA)
*        SEGMENT NAME=ABSIGHT, POINTER=(LPARNT,TWIN,LTWIN),
*        PARENT=((AREA),
*                  (BIRD,PHYSICAL,BIRCPDBD)), ...
FIELD    NAME=(BNAME,SEQ,M), ...
FIELD    NAME=DATE, ...
FIELD    NAME=REMARKS, ...

```

Fig. 20.16 DBD do PDB de AREA (esboço) — parelha virtual.

```

DBD      NAME=BIRDPDBD, ...
SEGMENT  NAME=BIRD, POINTER=TWIN, ...
*        LCHILD NAME=(ABSIGHT,AREAPDBD), PAIR=BASIGHT,
*        POINTER=SNGL
*        (FIELD statements for BIRD)
*        SEGMENT NAME=BASIGHT, POINTER=PAIRED,
*        PARENT=BIRD,
*        SOURCE=((ABSIGHT,,AREAPDBD))
FIELD    NAME=(AREA#,SEQ,M), ...
FIELD    NAME=DATE, ...
FIELD    NAME=REMARKS, ...

```

Fig. 20.17 DBD do PDB de BIRD (esboço) — parelha virtual.

A Fig. 20.16 mostra (veja os asteriscos) as modificações feitas no PDB de AREA (compare com a Fig. 20.12). *Não* foi incluída uma instrução LCHILD para o segmento BASIGHT. Também, a entrada POINTER de ABSIGHT não mais especifica PAIRED, mas sim requer a inclusão de um indicador lógico gêmeo de localização no prefixo. A ordenação desta cadeia lógica gêmea está especificada no *outro* DBD; veja abaixo. (Também é possível requerer uma cadeia lógica gêmea bidirecional, especificando LTWINBWD.) Observe que ali não há menção ao indicador de localização do pai físico, que o IMS provê automaticamente.

A Fig. 20.17 mostra as modificações feitas no DBD do PDB de BIRD (compare com a Fig. 20.13). A instrução LCHILD inclui a entrada adicional POINTER = SNGL, que é uma requisição para que cada ocorrência de BIRD inclua no seu prefixo um indicador de localização de filho lógico da primeira ocorrência de ABSIGHT na cadeia lógica gêmea correspondente a esta ocorrência BIRD. (Também é possível requerer que cada BIRD indique as localizações dos seus primeiro e último filhos lógicos, especificando POINTER = DBLE.) A instrução SEGM do segmento virtual BASIGHT especifica (a) POINTER = PAIRED, (b) o pai físico (somente), (c) uma nova entrada — SOURCE — que faz referência ao segmento real correspondente; esta instrução não tem outras entradas (nem mesmo BYTES). Por fim, *não* há modificações nas instruções FIELD de BASIGHT; em particular, a primeira define a “sequência de BASIGHT”, isto é, a sequência lógica gêmea para ABSIGHT.

Os DBDs lógicos são os mesmos de antes.

20.8 UMA OBSERVAÇÃO SOBRE A ESTRUTURA DE ARMAZENAMENTO

Das seções anteriores, podemos ver que, embora o usuário veja um LDB como uma estrutura hierárquica — pelo menos no que concerne à recuperação — a representação armazenada básica é algo mais complexa. Ela é, de fato, uma rede, na medida em que certos segmentos, os pais lógicos, podem ser indicados por qualquer quantidade de indicadores de localização (de pais lógicos), podendo portanto participar em qualquer quantidade de ocorrências LDBR. Entretanto, como o IMS não suporta “relacionamentos lógicos n -direcionais” para qualquer valor de $n > 2$, ela fica essencialmente limitada (em sua complexidade máxima) ao que chamamos de “redes de duas entidades”. Nas redes da Seção 20.7, por exemplo, os dois tipos de entidades são naturalmente AREA e BIRD.⁹ O problema de fornecedores-peças-projetos (veja a Parte 2 deste livro) é um exemplo de situação na qual seria requerida uma “rede de três entidades”, e portanto teria que ser manuseada de forma algo indireta no IMS (veja o Exercício 20.7).

20.9 BANCOS DE DADOS LÓGICOS ENVOLVENDO UM ÚNICO BANCO DE DADOS FÍSICO

Na introdução deste capítulo, nós afirmamos que os segmentos constituintes de uma ocorrência LDBR são retirados de diversas ocorrências PDBR distintas e que, por sua vez, as ocorrências PDBR podem pertencer a um ou mais PDBs distintos. Como um exemplo envolvendo apenas um PDB, consideremos a estrutura hierárquica ilustrada pela Fig. 16.1 (o banco de dados de educação), na qual o segmento raiz COURSE (campos COURSE#, TITLE, DESCRIPTN) possui o segmento dependente PREREQ (campos COURSE#, TITLE). É claro que podemos evitar a redundância que esta estrutura de PDB apresenta, substituindo PREREQ por um “segmento indicador de localização”, de forma a que cada ocorrência de COURSE tenha um conjunto de ocorrências de segmentos indicadores de localização dependentes que indiquem os COURSEs apropriados. As Figs. 20.18 e 20.19 mostram, respectivamente, a forma convencional de se fazer essa representação no papel e

⁹ Mas veja o Exercício 24.5.

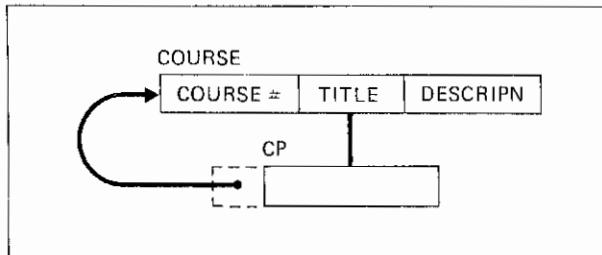


Fig. 20.18 PDB EDUC

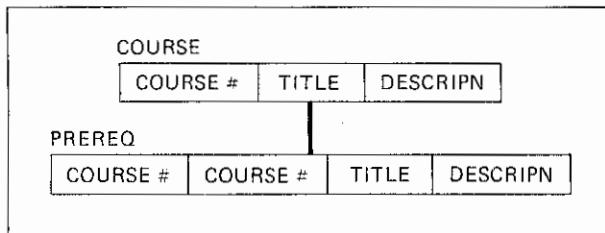


Fig. 20.19 LDB EDCP.

a estrutura hierárquica vista pelo usuário¹⁰ (para maior simplicidade, ignoramos os segmentos OFFERING, TEACHER, e STUDENT, bem como os dados de interseção que possam existir).

O segmento CP é ao mesmo tempo um filho lógico e um filho físico de COURSE. Inversamente, COURSE é ao mesmo tempo o pai físico e pai lógico de CP. Enfatizamos, entretanto, que nenhuma *ocorrência* CP tem a mesma *ocorrência* COURSE como pai físico e pai lógico ao mesmo tempo. O DBD do PDB de educação inclui uma instrução LCHILD para CP (em seguida à instrução SEGM de COURSE), e, além disso, a instrução SEGM de CP especifica COURSE como pai físico e como pai lógico.

O PDB de educação da Fig. 20.18 envolve um relacionamento lógico unidirecional (de CP para COURSE) que, como já vimos, nos possibilita encontrar todos os PREREQS de um determinado COURSE. Também é possível adicionar-se um relacionamento lógico

¹⁰ Com base no mesmo PDB, também é possível definir-se um LDB de três níveis, consistindo de COURSE (raiz), PREREQ1 (primeiro nível dependente), e PREREQ2 (segundo nível dependente), PREREQ1 representando os pré-requisitos imediatos de COURSE, e PREREQ2 representando os pré-requisitos imediatos de PREREQ1. De forma semelhante, podem ser definidos LDBs de quatro, cinco, ... níveis (até o máximo de 15 níveis). Na prática, esta é uma possibilidade muito útil, mas por simplicidade manteremos os dois níveis no nosso exemplo.

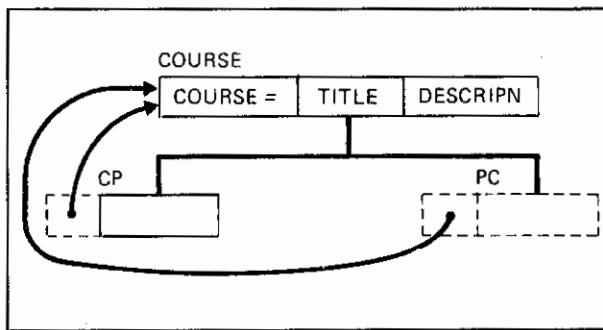


Fig. 20.20 O PDB EDUC (com parelha virtual) – representação convencional,

na outra direção (convertendo-o em um relacionamento bidirecional), de tal forma que, adicionalmente, podemos encontrar todos os COURSES que têm um determinado PREREQ. Isto é feito pela inclusão de um segundo tipo de segmento indicador de localização – digamos PC – fisicamente dependente de COURSE, ficando cada ocorrência COURSE com um conjunto de ocorrências PC que indicam as localizações das ocorrências COURSE das quais o curso original é um pré-requisito. PC e CP são segmentos parelhos. Se supusermos o uso de parelha virtual (e que PC seja o membro virtual do par), as Figs. 20.20, 20.21, e 20.22 mostram, respectivamente, a forma convencional de se representar essa situação, as estruturas existentes, e os dois LDBs que são vistos pelo usuário.

Enfatizamos novamente que nenhuma *ocorrência* filho terá a mesma *ocorrência* de segmento tanto como pai físico quanto como pai lógico. Os dois indicadores de localização de CP para COURSE na Fig. 20.21 são, respectivamente, um indicador de localização do pai lógico e um indicador de localização do pai físico; em uma ocorrência CP eles indicarão as localizações de ocorrências *diferentes* de COURSES. Com referência à Fig. 20.22, o usuário emprega o LDB EDCP para encontrar os PREREQS para um determinado COURSE, e o LDB EDPC para encontrar os COURSES para um determinado PREREQ.

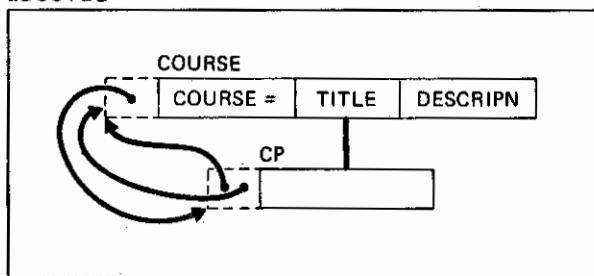
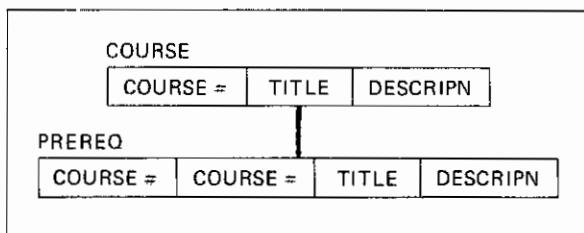


Fig. 20.21 O PDB EDUC (com parelha virtual).

(Os nomes dos segmentos nesses LDBs são naturalmente arbitrários; o ponto importante é que EDCP tem que ser definido em termos do filho lógico CP, e EDPC tem que ser definido em termos do filho lógico PC.)

Veja os Exercícios 20.3, 20.4, e 20.5.

EDCP LDB



EDPC LDB

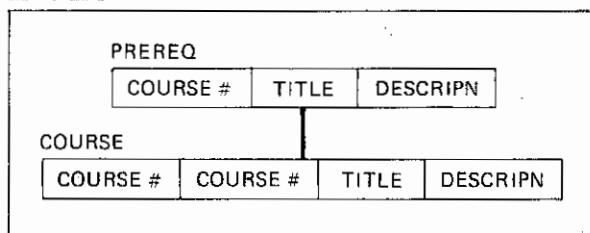


Fig. 20.22 Os LDBs EDCP e EDPC.

20.10 ALGUMAS REGRAS E RESTRIÇÕES

A definição de um LDB está sujeita a uma série de regras e restrições, documentadas em detalhes em [19.1]. Nesta Seção examinaremos algumas dessas regras, provavelmente as mais importantes do ponto de vista prático. Deixamos como exercício para o leitor o desenvolvimento de diagramas adequados para ilustrar estes pontos.

1. A raiz de um LDB tem que ser também a raiz de um PDB.
2. Um segmento filho lógico tem que ter um pai físico e um pai lógico. Segue-se daí que (a) uma raiz não pode ser um filho lógico; (b) o IMS não pode suportar diretamente "relacionamentos n -direcionais" para $n > 2$.
3. Um filho físico de um filho lógico pode aparecer como dependente do segmento concatenado (filho-lógico/pai-lógico) no LDB. Por exemplo, consideremos os bancos de dados da Seção 20.2. Suponhamos que no PDB AREA (Fig. 20.3) REMARKS consista de uma seqüência de caracteres de comprimento variável entre 100 e 1000 caracteres. Ao invés de se incluir REMARKS como um campo dentro de SIGHTING, poderia ter sido preferível nessa situação ter um segmento REMARKS fisicamente dependente do seg-

mento SIGHTING (se o segmento REMARKS for de 100 *bytes*, haveria de 1 a 10 ocorrências deste para cada ocorrência SIGHTING). O LDB SURVEY (Fig. 20.4) poderia então consistir de três níveis, com AREA no nível raiz, o segmento concatenado SIGHTING no segundo nível, e REMARKS no terceiro nível. Isto é um exemplo de como os “dados variáveis de interseção” são manuseados no IMS (“variáveis” aqui significando “comprimento variável”). Incidentalmente, o segmento REMARKS provavelmente não incluiria um campo de ordenação (veja o final da Seção 16.2).

4. Um filho físico de um pai lógico pode aparecer como um dependente do segmento concatenado (filho-lógico/pai-lógico) no LDB, desde que o pai lógico seja o primeiro pai lógico encontrado na definição do caminho hierárquico do LDB.¹¹ Continuando com o exemplo da Seção 20.2, suponhamos que BDESCN seja representado como um segmento físico dependente de BIRD (tal como REMARKS, podendo consistir de uma seqüência de caracteres de comprimento altamente variável). BDESCN poderá então aparecer como um filho do segmento concatenado SIGHTING no LDB SURVEY. (Além disso, se BDESCN tiver um filho físico chamado, digamos, SUBDESCN, então SUBDESCN poderá aparecer como um filho de BDESCN no LDB SURVEY.) Entretanto, se BIRD tiver também um filho físico — digamos X — que por sua vez tenha um pai lógico Y tal que X seja tanto um filho físico quanto um filho lógico, então, embora a concatenação de X e Y pudesse ser incluída em SURVEY (como filho do segmento SIGHTING), nenhum filho físico de Y poderia ser incluído assim.

5. O pai físico de um pai lógico pode aparecer como um dependente do segmento concatenado (filho-lógico/pai-lógico) no LDB. Suponhamos que BIRD tenha um segmento pai físico GENUS. Então GENUS poderá ser visto como um *filho* do segmento concatenado SIGHTING no LDB SURVEY. Além disso, se GENUS por sua vez tiver um segmento pai físico FAMILY, então FAMILY poderá ser visto como um filho de GENUS no LDB SURVEY, e assim por diante, até 15 níveis (sempre o limite global no IMS). Em outras palavras, é possível inverter-se a estrutura hierárquica do PDB, pelo menos até certo ponto. Para maiores detalhes, o leitor deverá procurar a referência [19.1].

6. Se X e Y são dois segmentos no caminho hierárquico de um LDB, todos os segmentos que se encontram no caminho entre X e Y no PDB básico têm também que ser incluídos (nas mesmas posições relativas) naquele caminho hierárquico do LDB.

20.11 RESUMO

O dispositivo de bancos de dados lógicos do IMS torna possível superar alguns dos problemas de “complexidade para recuperação” mencionados na seção sobre hierarquias (Seção 3.3) do Capítulo 3. Ele fornece um método para redução da redundância nos dados armazenados, permitindo ao mesmo tempo redundância na visão do usuário onde isto for desejável. Mais importante, talvez, seja a possibilidade de permitir que usuários vejam os dados de muitas maneiras diferentes (embora a visão individual de cada usuário seja ainda hierárquica). Esses objetivos são alcançados por meio de uma estrutura de armazenamento estendida que é, efetivamente, uma rede (limitada), embora voltemos a enfatizar que o usuário não vê uma estrutura em rede. Concluindo, devemos mencionar que os bancos de

11

Esta restrição é relaxada para um LDB definido em termos de um único PDB, como descrito na nota 9 de rodapé.

dados lógicos fornecem uma reestruturação essencialmente *estática* dos dados. Isto não tem o significado da criação *dinâmica* de uma estrutura que não seja previamente conhecida pelo sistema, como foi feito em muitos exemplos nas Partes 1 e 2 deste livro – veja o exemplo 3.5.4.

EXERCÍCIOS

20.1 Reestruture o banco de dados de publicações do Exercício 16.1 em dois PDBs, de forma que possam ser definidos LDBs especificamente talhados para responder às duas seguintes solicitações:

Encontre os autores de uma determinada publicação.

Encontre todas as publicações de um determinado autor. Que LDBs irá o usuário ver no seu projeto? Escreva todos os DBDs relevantes (em esboço); para maior simplicidade, pode ser considerado que os dois PDBs usam organização direta hierárquica.

20.2 Considerando os dados de amostragem da Fig. 20.5, desenhe um diagrama da estrutura de armazenamento (mostrando todos os indicadores de localização relevantes) correspondentes ao seguinte:

- O LDB SURVEY (Figs. 20.7, 20.8, e 20.9).
- Os LDBs ABSURVEY e BASURVEY com parelha física (Figs. 20.12 e 20.13).
- Os LDBs ABSURVEY e BASURVEY com parelha virtual (Figs. 20.16 e 20.17).

20.3 Monte todas as ocorrências de segmentos juntamente com seus indicadores de localização dos pais lógicos para o PDB EDUC (Fig. 20.18), usando os dados de amostragem da Fig. 20.23.

COURSE =	PREREQ COURSE =s
L02	—
M16	—
M19	—
M23	M16, M19
M27	L02
M30	L02, M23, M27

Fig. 20.23 Dados de amostragem do PDB EDUC.

20.4 Monte todas as ocorrências de segmentos juntamente com os indicadores de localização dos seus pais lógicos, filhos lógicos, e pais físicos para o PDB EDUC (Fig. 20.21), considerando os mesmos dados de amostragem do exercício anterior.

20.5 Defina a nível razoável de detalhes os DBDs para o PDB EDUC da Fig. 20.21 e os LDBs da Fig. 20.22.

20.6 Como você manusearia o problema de fornecedores-peças-projetos em IMS?

20.7 O que está envolvido na carga do PDB EDUC da Fig. 20.21?

20.8 O relacionamento lógico bidirecional da Seção 20.7 pode ser implementado de três formas: via parelha física, via parelha virtual com ABSIGHT virtual, ou via parelha virtual com BASIGHT virtual. Até que ponto pode o DBA mudar de uma para outra dessas três técnicas sem afetar os usuários existentes?

REFERÊNCIAS E BIBLIOGRAFIA

Veja [16.1] e [19.1].

21

Índices Secundários do IMS

21.1 INTRODUÇÃO

Como explicado no Capítulo 19, a seqüência das raízes em um banco de dados HISAM ou HIDAM é suportada por meio de um *índice* no campo de ordenação do segmento raiz. Para HISAM, a indexação é fornecida pelo método de acesso padrão do OS/VS (ISAM ou VSAM); para HIDAM, o índice é por si mesmo um banco de dados IMS, tendo por método de acesso básico VSAM ou uma combinação de ISAM e OSAM. Como esses índices estão no campo de ordenação do segmento raiz, podemos nos referir a eles como índices *primários* (o campo de ordenação do segmento raiz, que tem que ter valores únicos em HISAM e HIDAM, pode ser visto como a chave primária do registro do banco de dados físico); e a ordenação correspondente das raízes, ou seja, por valores crescentes do campo de ordenação, pode ser tomada como a seqüência primária de processamento. Neste capítulo apresentaremos uma introdução à indexação *secundária* do IMS, e a noção associada de seqüência secundária de processamento.

No Capítulo 2 nós definimos um índice secundário como um índice em um campo outro que não a chave primária. Naquele momento, entretanto, nós estávamos supondo tacitamente que o arquivo a ser indexado não era mais complexo do que um arquivo sequencial convencional. Em IMS, naturalmente, o arquivo (banco de dados) a ser indexado é hierárquico em estrutura, e o conceito de indexação secundária tem que ser estendido de acordo. De fato, um índice secundário em IMS pode ser usado:

- para indexar um determinado segmento, raiz ou dependente, com base em qualquer campo daquele segmento;
- para indexar um determinado segmento, raiz ou dependente, com base em qualquer campo de um dependente (não necessariamente um filho imediato) daquele segmento.¹

¹ Por “dependente daquele segmento” aqui queremos dizer um dependente do segmento no seu banco de dados *físico*.

Em todos os casos, o “campo” no qual o índice é baseado pode ser uma concatenação de até cinco campos, não necessariamente contíguos, tomados em qualquer ordem. O índice é implementado como um banco de dados INDEX, de forma muito parecida com o índice primário em HIDAM, exceto que o método de acesso básico para este banco de dados INDEX tem que ser VSAM, e não ISAM/OSAM. O banco de dados a ser indexado tem que estar em HISAM, HDAM, ou HIDAM, não em HSAM (não podendo este ser ele próprio um INDEX). Também é possível a indexação de um banco de dados lógico, mas detalhes sobre esse assunto fogem ao escopo deste livro.

Para ilustrar as possibilidades, consideraremos novamente o banco de dados de educação da Fig. 16.1 (novamente mostrado aqui como Fig. 21.1).

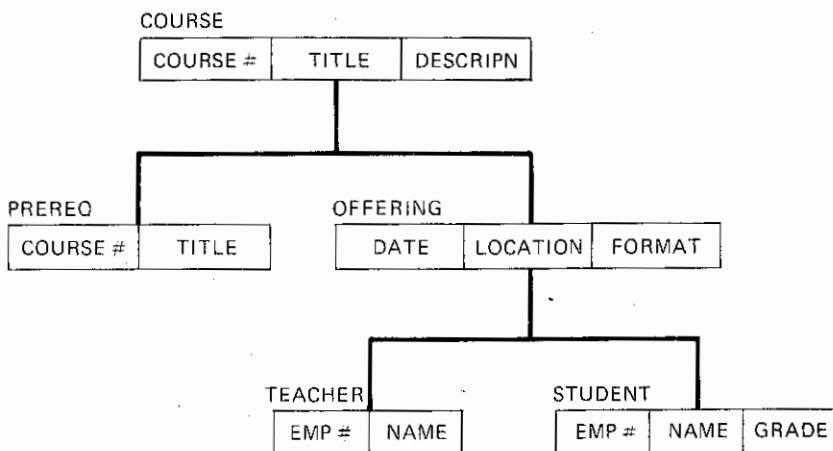


Fig. 21.1 Estrutura do banco de dados de educação.

A lista a seguir esboça alguns dos muitos índices possíveis de serem colocados neste banco de dados.

- Um índice para COURSEs no campo TITLE
- Um índice para COURSEs no campo LOCATION do segmento OFFERING
- Um índice para OFFERINGS no campo LOCATION
- Um índice para OFFERINGS no campo EMP# do segmento TEACHER

Vamos agora analisar cada um destes quatro exemplos em detalhes.

21.2 INDEXANDO A RAIZ EM UM CAMPO QUE NÃO O CAMPO DE SEQÜÊNCIA

Como primeiro exemplo, consideremos um índice de COURSEs baseado nos valores do campo TITLE. Estaremos supondo inicialmente que os valores de TITLE são únicos. Para o estabelecimento de *qualquer* índice secundário, precisamos de um DBD para aquele índice – muito parecido com o DBD do banco de dados INDEX em HIDAM – juntamente

com entradas no DBD do banco de dados sendo indexado para indicar o(s) campo(s) no(s) qual(qual) o índice será baseado, e outros detalhes. O índice será então automaticamente construído pelo IMS quando da carga do banco de dados, e será automaticamente mantido quando o banco de dados for atualizado posteriormente. No caso presente, nós precisamos das duas seguintes instruções no DBD do banco de dados de educação:

```
LCHILD NAME=(TPTR,TXDBD),POINTER=INDX  
XDFLD NAME=XTITLE,SRCH=TITLE
```

Essas duas instruções têm que aparecer juntas, como mostrado. Elas podem aparecer em qualquer lugar após a instrução SEGMENT de COURSE (e antes da próxima instrução SEGMENT), com a exceção de não poderem preceder a instrução FIELD do campo de ordenação (COURSE#), e, se o banco de dados for HIDAM, elas não poderão preceder a instrução LCHILD que conecte COURSE ao índice primário (veja a Seção 19.8).

A instrução LCHILD especifica que o segmento COURSE está indexado pelo segmento T PTR no banco de dados definido no DBD chamado TXDBD. Como mostramos abaixo, TXDBD é o DBD que define o banco de dados índice secundário, e T PTR é o "segmento índice" — isto é, é o (único) tipo de segmento no banco de dados índice. A entrada "POINTER = INDIX" especifica que a ligação entre COURSE e T PTR é realmente uma conexão por índice, e não uma conexão filho-lógico/pai-lógico. A instrução XDFLD ("campo indexado") especifica o campo no qual deve ser construído o índice secundário. No exemplo, este campo é TITLE — veja a entrada SRCH. Entretanto, uma SSA que inclua uma condição sobre o campo de nome TITLE — por exemplo, TITLE = 'DYNAMICS' — não fará com que o IMS use o índice na pesquisa da ocorrência desejada do segmento. Na verdade, o usuário tem que *forçar* o uso do índice (quando desejado) usando o nome de campo XTITLE, especificado na entrada NAME da instrução XDFLD. Para forçar o IMS a usar o índice secundário na pesquisa de um curso com o TITLE, por exemplo, de 'DYNAMICS', o usuário escreveria

```
GU COURSE (XTITLE='DYNAMICS')
```

Este GU faz com que o IMS obtenha o segmento índice 'DYNAMICS' e então obtenha o segmento COURSE indicado por este índice. Observe que o segmento recuperado (na área de I/O) é exatamente o usual — especificamente, não inclui um campo extra correspondente a XTITLE.

Vejamos agora o DBD do índice secundário.

```
DBD      NAME=TXDBD,ACCESS=INDEX  
SEGMENT  NAME=TPTR,BYTES=33  
FIELD    NAME=(TITLE,SEQ),BYTES=33,START=1  
LCHILD   NAME=(COURSE,EDUCPDBD),INDEX=XTITLE
```

Como já mencionado anteriormente, o DBD de um índice secundário é muito semelhante ao de um índice primário em HIDAM (veja a Seção 19.8 para detalhes). A única diferença significativa no exemplo acima é que a entrada INDEX na instrução LCHILD se refere a XTITLE ao invés de a TITLE, isto é, ao "campo XD" ao invés de ao campo original. Mais adiante discutiremos algumas outras possíveis diferenças.

O índice secundário de COURSE que definimos conterá uma ocorrência de índice para cada ocorrência de COURSE. Cada segmento de índice conterá um valor de TITLE e um indicador da localização do curso correspondente. Dado este índice, o usuário pode agora escolher uma verificação do banco de dados em uma *seqüência secundária de processamento*, ou seja, por valores ascendentes do campo de título de curso. (A seqüência de processamento *primário* é por valores crescentes do campo de número de curso.) Para especificar que esta seqüência secundária é requerida, basta que o usuário inclua a entrada

```
PROCSEQ=TXDBD
```

na instrução PCB do PCB (veja a Seção 17.2). As operações “get unique” e “get next” usando este PCB operarão agora em termos desta seqüência; observe, entretanto, que se essas operações incluírem uma SSA envolvendo uma condição no campo de título, a condição deverá ser expressa em termos de XTITLE e não de TITLE.

Se o PCB não incluir uma entrada PROCSEQ, a seqüência *primária* de processamento será usada. Ainda nesse caso as SSAs podem incluir referências aos “campos XD” tais como XTITLE, mas normalmente não deverão por razões de eficiência. Por exemplo, a operação

```
GU COURSE (XTITLE='DYNAMICS')
```

será implementada como se segue, se for usada a seqüência primária de processamento:

```
get to start of primary sequence
loop: get next COURSE in primary sequence
      get index segment for DYNAMICS
      if index segment points to current COURSE, go to exit
      go to loop
```

A razão é que o “get unique” (que é realmente um “get first”) tem que localizar o primeiro COURSE que satisfaça à SSA de acordo com a seqüência *primária*. Portanto o IMS pesquisa os COURSEs nessa seqüência, testando um a um contra a SSA; e como a SSA envolve um campo XD, cada um desses testes provoca um acesso ao índice secundário. O uso de TITLE ao invés de XTITLE significaria que cada um desses testes poderia ser feito diretamente sobre os dados do segmento COURSE, sem envolver qualquer acesso a índices.

Observações semelhantes a essas acima aplicam-se ao uso de um campo XD em uma SSA conjugada com uma seqüência *secundária* de processamento, se a seqüência secundária não for a associada àquele campo XD. Por exemplo, o “get unique” do exemplo acima (usando XTITLE) resultaria em mau desempenho, se a seqüência de processamento fosse baseada em valores do campo DESCIPN. Por isso uma única SSA não deve normalmente incluir referências a dois campos XD distintos, pois pelo menos um deles terá que deixar de corresponder à seqüência de processamento. Há algumas situações em que pode ser vantajoso o uso de um índice secundário em conjunção com uma seqüência de processamento não baseada naquele índice, mas os detalhes estão além do escopo deste livro; para maiores informações, veja [19.1]. A partir de agora, restringiremos a nossa atenção ao caso em que a seqüência de processamento é a associada ao campo XD envolvido.

Relacionamos abaixo alguns pontos adicionais que necessitam atenção do usuário ao usar seqüência de processamento secundária.

- Os valores do campo indexado não têm que ser únicos. Na nossa ilustração, por exemplo, os cursos não têm que ter títulos únicos. Nesse caso, a instrução FIELD no DBD para o índice (TXDBD no exemplo) tem que incluir a especificação M (múltiplo) na entrada NAME:

```
FIELD NAME=(TITLE,SEQ,M),...
```

Se n cursos tiverem o mesmo título, haverá n segmentos índice para aquele título no banco de dados INDEX; isto é, cada ocorrência de COURSE terá uma entrada de índice indicando aquele título. Veja em [19.1] detalhes de como as "duplicatas" são ordenadas com relação à seqüência secundária de processamento.

- A chave totalmente concatenada que foi retornada na área de recebimento da chave (no PCB) conterá um valor TITLE ao invés de um valor COURSE#. De fato, o campo indexado está sendo tratado como o campo de ordenação dos segmentos COURSE.
- Não obstante os itens anteriores, o usuário tem possibilidade de atualizar os campos TITLE (usando REPL) e não tem possibilidade de atualizar os campos COURSE#. Em outras palavras, as regras para REPL não são afetadas pelo uso de uma indexação secundária. Observe que a atualização do campo TITLE de um determinado COURSE pode fazer com que aquele COURSE mude de posição na seqüência secundária; por exemplo, a mudança do título do curso M19 de 'CALCULUS' para 'INTEGRAL CALCULUS' irá movê-lo antes de 'DYNAMICS' para depois de 'DYNAMICS'. (O IMS automaticamente faz com que todos os índices secundários reflitam tais atualizações, removendo o segmento de índice antigo e criando um novo.) Assim, uma operação subsequente de "get next" poderá recuperar novamente o mesmo COURSE.
- O "segmento alvo do índice" (isto é, o segmento indicado pelo índice – COURSE no nosso exemplo) não pode ser objeto de operações ISRT ou DLET quando o processamento for via seqüência secundária. Se o alvo do índice for um dependente ao invés de uma raiz, esta restrição se aplica não somente a ele, mas também a todos os seus ancestrais (veja as Seções 21.4 e 21.5).
- Se o banco de dados a ser indexado residir em HISAM ao invés de HIDAM ou HDAM, os indicadores de localização no índice terão que ser simbólicos e não diretos (como a situação com indicadores de localização de pais lógicos quando o pai lógico está em HISAM, conforme foi descrito no Capítulo 20). Neste caso, a entrada 'POINTER = INDX' na instrução LCHILD do banco de dados a ser indexado tem que ser substituída por 'POINTER = SYMB' e 'POINTER = SYMB' tem que ser também especificado na instrução LCHILD do próprio banco de dados INDEX.
- Não podem ser definidos índices secundários sobre bancos de dados HISAM que incluam quaisquer grupos de arquivos secundários.

Os itens mencionados acima, adequadamente modificados onde for necessário, aplicam-se sempre que for usada uma seqüência secundária de processamento, e não somente ao caso particular considerado nesta seção.

21.3 INDEXANDO A RAIZ EM UM CAMPO EM UM DEPENDENTE

Suponhamos que queremos encontrar os números de cursos para todos os COURSEs que possuem uma oferta em Stockholm. A seguinte codificação (usando uma chamada de caminho) representa uma possível solução ao problema.

```
NC GN      COURSE+D
        OFFERING (LOCATION='STOCKHOLM')
if not found, go to exit
add COURSE# to result list
go to NC
```

(supondo que estamos inicialmente posicionados no início do banco de dados. Também ignoramos a questão da eliminação dos números de cursos em duplicata do resultado.) Entretanto, esta codificação não é particularmente eficiente, pois consiste essencialmente de uma varredura seqüencial de todo o banco de dados (embora alguns segmentos possam ser saltados se o banco de dados armazenado contiver indicadores de localização filho/gêmeo). Além disso, o usuário é forçado a recuperar segmentos OFFERING, embora estes não sejam realmente desejados.

Uma solução mais eficiente do problema pode ser obtida usando-se um índice secundário, indexando os segmentos COURSE na base de valores LOCATION – isto é, indexando a raiz em um campo em um dependente. Os DBDs necessários são basicamente como os da Seção 21.2, com exceção de: (a) para o segmento COURSE, a instrução XDFLD é

```
XDFLD NAME=XLOC, SRCH=LOCATION, SEGMENT=OFFERING
```

(especificando que o campo XD (XLOC) deriva-se do campo LOCATION do segmento OFFERING); e (b) no DBD do banco de dados INDEX, a instrução LCHILD é

```
LCHILD NAME=(COURSE, EDUCPDBD), INDEX=XLOC
```

Incidentalmente poderíamos ter dado ao campo XD o nome de LOCATION sem risco de ambigüidade, mas para maior clareza escolhemos o nome distinto XLOC.

Os índices secundários conterão um segmento índice para cada ocorrência de OFFERING no banco de dados de educação. Cada um desses segmentos índice indicará a ocorrência de COURSE correspondente (o pai da OFFERING envolvida). Se o banco de dados de educação contiver m COURSEs e em média n OFFERINGS por COURSE, o índice conterá mn segmentos índice, com uma média de n segmentos índice indicando um COURSE qualquer. A seqüência de processamento secundária do banco de dados de educação fica definida pela seqüência desses segmentos índice, isto é, pelos valores ascendentes de LOCATION (novamente, veja [19.1] sobre detalhes da ordenação das duplicatas). Observe que, em média, cada COURSE individual (juntamente com todos os seus dependentes) aparecerá n vezes nesta seqüência, e que em geral *não* aparecerão juntas todas as ocorrências de um determinado COURSE. Em outras palavras, quando visto via índices secundários, o banco de dados parece conter n vezes a quantidade de registros do banco de dados – embora esses registros não sejam todos independentes uns dos outros; por exemplo, a mudança do título de uma determinada ocorrência de COURSE provocará a mesma mudança em todas as outras ocorrências de COURSE do curso envolvido.

Supondo que o PCB especifique a seqüência secundária de processamento (PROC-SEQ = nome-do-DBD-do-índice), podemos então escrever

```
NC GN      COURSE (XLOC='STOCKHOLM')
  if not found, go to exit
  add COURSE# to result list
  go to NC
```

(Observe que a sintaxe faz parecer que XLOC é um campo dentro do segmento COURSE.) Cada interação desta codificação faz com que o IMS obtenha o próximo segmento índice de 'STOCKHOLM' e depois obtenha o segmento de COURSE indicado por este índice. É portanto mais eficiente do que a codificação anteriormente dada, pois basicamente envolve somente $2N$ acessos ao banco de dados (onde N é o número de ofertas em Stockholm) ao invés de uma varredura de todo o banco de dados. No entanto, novamente ignoramos o problema da eliminação dos números de cursos duplicados.

Observe que a chave totalmente concatenada retornada na área de recebimento da chave quando é usada esta seqüência secundária conterá um valor de LOCATION no lugar de um valor de COURSE# (verdadeiro para todos os tipos de segmentos na hierarquia).

O AND Independente

Quando um segmento é indexado com base em um campo de algum dependente daquele segmento, torna-se possível escrever-se uma SSA para aquele segmento envolvendo duas ou mais condições no campo XD, separadas pelo operador especial # ("AND independente") – por exemplo,

```
GU      COURSE (XLOC='STOCKHOLM'#XLOC='OSLO')
```

Para entender o significado, o leitor deve perceber primeiramente que, para uma dada ocorrência do segmento indexado (COURSE no exemplo) existe realmente um *conjunto* de valores do campo XD (XLOC no exemplo). Uma SSA como esta acima é considerada como satisfeita se, para cada uma das condições distintas separadas por operadores #, existir pelo menos um valor no conjunto para o qual a condição seja verdadeira. No exemplo, portanto, o GU irá recuperar o primeiro COURSE que apresenta tanto uma oferta em Stockholm como uma oferta em Oslo – isto é, primeiro na seqüência de processamento especificada.

Observe que se # fosse substituído pelo AND ordinário ("dependente"), a SSA estaria solicitando um COURSE com alguma oferta *única* tanto em Stockholm quanto em Oslo. O IMS não reconheceria nenhuma contradição na solicitação, mas pesquisaria o banco de dados e eventualmente retornaria um código de status "não encontrado".

Como outro exemplo, suponhamos que os COURSES foram indexados com base nos valores de DATE do segmento OFFERING, e consideremos a seguinte operação:

```
GU      COURSE (XDATE>'691231'#XDATE<'710101')
```

(onde XDATE é o nome do campo XD). Como está escrita, esta operação irá recuperar o primeiro curso com uma oferta posterior a 1969 e uma oferta – não necessariamente a mesma – anterior a 1971. Se o AND independente fosse substituído pelo AND dependente, a operação iria recuperar o primeiro curso com uma oferta em 1970.

21.4 INDEXANDO UM DEPENDENTE EM UM CAMPO DAQUELE DEPENDENTE

Como mencionado na Seção 21.1, o segmento alvo do índice – o segmento indicado pelo índice – não tem que ser uma raiz. Entretanto, se não for, o efeito é o de reestruturar a hierarquia de forma que ele se *torne* a raiz na estrutura vista pelo usuário.² Como um exemplo, consideremos o caso de indexação das OFFERINGS com base nos valores do campo LOCATION.

Novamente os DBDs são essencialmente os mesmos que na Seção 21.2. O DBD de educação incluirá instruções LCHILD e XDFLD apropriadas para OFFERING; o DBD do índice fará referência ao nome apropriado do XD (digamos XLOC; *não podemos* usar o nome LOCATION desta vez, porque o campo genuíno e o campo XD têm que ter nomes únicos no conjunto de campos de OFFERING). Observe novamente que, sintaticamente, os campos XD são sempre considerados como campos do segmento alvo. O índice irá conter um segmento índice para cada ocorrência de OFFERING do banco de dados de educação; de fato, ele é exatamente o mesmo índice da seção 21.3, exceto que os segmentos índice indicam OFFERINGS ao invés de COURSES.

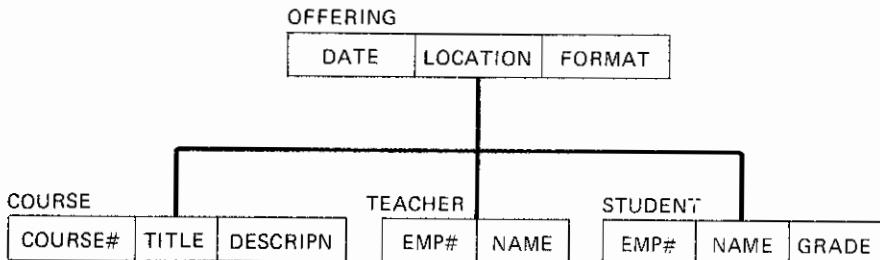


Fig. 21.2 Indexando OFFERINGS em LOCATION: Estrutura secundária de dados.

A Fig. 21.2 mostra a *estrutura secundária de dados* resultante quando o usuário especifica esta seqüência secundária de processamento no PCB. A Fig. 21.3 mostra o PCB correspondente (somente os detalhes relevantes; LXDBD é o nome do DBD do índice de LOCATION).

```
PCB      TYPE=DB, . . . , KEYLEN=12, PROCSEQ=LXDBD
SENSEG  NAME=OFFERING
SENSEG  NAME=COURSE, PARENT=OFFERING
SENSEG  NAME=TEACHER, PARENT=OFFERING
SENSEG  NAME=STUDENT, PARENT=OFFERING
```

Fig. 21.3 PCB da estrutura secundária de dados (Fig. 21.2).

² Como sempre, estamos supondo o uso da seqüência secundária de processamento. Como já mencionado antes, podem ocorrer situações em que os índices secundários sejam usados vantajosamente com a seqüência primária de processamento; nessa situação, não ocorre a reestruturação da hierarquia.

As regras para a definição de uma estrutura secundária de dados são as seguintes.

- O segmento alvo do índice torna-se a raiz.
 - Os ancestrais do segmento alvo do índice tornam-se os dependentes mais à esquerda desta raiz, em ordem inversa. (Se COURSE tivesse um pai CATEGORY no banco de dados de educação, CATEGORY seria um *dependente* de COURSE na estrutura secundária.)
 - Os dependentes do segmento alvo do índice aparecem exatamente como no banco de dados original, exceto que eles ficam à direita do(s) segmento(s) dependente(s) introduzido(s), como explicado no parágrafo anterior.
 - Não há outros segmentos incluídos. (No exemplo, a estrutura secundária não inclui PREREQs.)
- O PCB da Fig. 21.3 define uma estrutura de acordo com estas regras.

Há algumas semelhanças entre uma estrutura de dados como a da Fig. 21.2 e um banco de dados lógico; em particular, convidamos o leitor a comparar as regras anteriores com as observações da Seção 20.10. As operações "get unique" e "get next" funcionam em termos da estrutura secundária (e seqüência secundária). As chaves totalmente concatenadas (da estrutura da Fig. 21.2) consistem de um valor de LOCATION seguido, se apropriado, por um valor COURSE# ou EMP#. Podem ser usadas operações ISRT/DLET para TEACHER e STUDENT, mas não para OFFERING ou COURSE.

Como uma ilustração sobre o possível uso de uma estrutura como essa, vamos estender nosso exemplo de cursos em Stockholm como se segue. Suponhamos que queremos encontrar não somente os números dos cursos, mas também os números de empregados dos professores correspondentes, para todos os cursos que possuem uma oferta em Stockholm. Para simplificar, suponhamos que cada oferta possui exatamente um professor. Em outras palavras, queremos produzir um relatório contendo uma linha para cada oferta em Stockholm, dando o número do curso e o número do professor (e possivelmente também outras informações, como a data da oferta). O índice da Seção 21.3 não ajuda muito aqui; ele nos permite encontrar um curso que se qualifica, mas não nos dá meios imediatos para saber qual dos diversos professores dependentes é aquele que queremos. Com a estrutura da Fig. 21.2 podemos escrever

```
NC GN      OFFERING (XLOC='STOCKHOLM')
                  COURSE
                  if not found, go to exit
                  GN      TEACHER
                  print COURSE# and teacher EMP#
                  go to NC
```

O ponto é que, uma vez que tenhamos encontrado uma oferta em Stockholm, sabemos que tem que haver exatamente um curso correspondente e um professor correspondente, e esses são os que desejamos.

Incidentalmente, é interessante observar como uma pequena alteração na colocação do problema (encontre também os professores) leva a uma modificação grande na solução (torna-se desejável uma hierarquia reestruturada, com a consequência de ser necessária também uma revisão no procedimento de acesso). Em outras palavras, uma pequena alteração na consulta leva a grandes alterações na solução. Este efeito perturbador

pode ser atribuído ao fato de que os índices secundários representam uma tentativa de fornecer simetria de acesso (o acesso ao banco de dados via um campo indexado devendo ser semelhante ao acesso via “chave primária”, isto é, o campo de ordenação da raiz), enquanto que uma hierarquia é uma estrutura fundamentalmente assimétrica (o fornecimento de acesso via um campo indexado portanto requer uma reestruturação, de forma que o campo indexado *se torne* o campo de ordenação da raiz).

21.5 INDEXANDO UM DEPENDENTE EM UM CAMPO EM UM DEPENDENTE DE NÍVEL MAIS BAIXO

Esta, a última das quatro possíveis combinações, não ilustra realmente novos pontos, mas a incluímos para que o trabalho ficasse completo. Como exemplo, vamos considerar o caso de indexar OFFERINGS com base nos números de empregados de TEACHER.

Novamente, os DBDs são essencialmente os mesmos de antes. O DBD de educação incluirá instruções LCHILD e XDFLD apropriadas para OFFERING; a instrução XDFLD tem que incluir a entrada “SEGMENT = TEACHER” (pois o campo XD é derivado de um campo do segmento TEACHER, não do segmento OFFERING). O DBD de índice fará referência ao nome do campo XD (digamos XEMP#). O índice irá conter um segmento índice para cada ocorrência de TEACHER, e cada um desses segmentos índice indicará a OFFERING correspondente. O usuário verá a mesma estrutura secundária da Fig. 21.2; entretanto, pode-se agora ter acesso às OFFERINGS via o campo XD XEMP#, sendo a ordenação e as chaves totalmente concatenadas definidas em termos de XEMP#. Vamos apresentar apenas um exemplo codificado. O problema é: “Encontre a data de todas as ofertas ministradas pelo empregado número 876225 em Stockholm.”

```
NO GN      OFFERING (LOCATION='STOCKHOLM'&XEMP#='876225')
  if not found, go to exit
  print DATE
  go to NO
```

21.6 DISPOSITIVOS ADICIONAIS

O dispositivo de indexação secundária do IMS apresenta outras possibilidades além das que vimos até agora. Vamos esboçar abaixo duas delas. Para informações sobre as demais – dados do usuário no índice, uso de campos subsequentes, bancos de dados de índices compartilhados e o uso de “campos relacionados ao sistema” o leitor deve procurar a referência [19.1].

- Indexação dispersa

É possível suprimir-se a criação de segmentos índice para valores determinados do campo XD (veja a discussão sobre indexação seletiva no Capítulo 2). Parte-se do princípio de que os valores suprimidos não são de interesse para aquele índice; se ocorrer uma tentativa para encontrar um segmento via uma SSA especificando um valor suprimido do campo XD, esta causará uma condição de “não encontrado”, mesmo que exista o segmento qualificado. Em outras palavras, os segmentos não indexados simplesmente não aparecem na seqüência secundária de processamento.

- Dados duplicados

Um índice secundário é por si mesmo um banco de dados, e pode ser processado independentemente do banco de dados que indexa. Para aumentar a utilidade deste dispositivo, o administrador de banco de dados pode especificar que certos campos do segmento fonte do índice (o segmento de onde são derivados os valores do campo XD) sejam duplicados no segmento correspondente no índice. O IMS irá automaticamente manter esses campos duplicados quando for feita uma atualização no segmento fonte. Uma aplicação em que elevado desempenho seja uma necessidade e que requeira somente dados que tenham sido duplicados no índice podem usar o índice no lugar do banco de dados original.

21.7 RESUMO

Podemos resumir os principais itens introduzidos neste capítulo como se segue:

- De forma geral, um índice secundário em IMS pode ser usado para indexar um determinado segmento com base em qualquer campo daquele segmento ou de qualquer dependente físico daquele segmento. (Há algumas exceções a esta afirmativa: certos tipos de segmentos, como por exemplo segmentos filhos lógicos, não podem ser indexados; veja detalhes em [19.1].) O campo ou combinação de campos sobre o qual o índice é montado é representado por um “campo XD”, que aparece como um campo adicional do segmento alvo do índice. As SSAs têm que ser expressas em termos deste campo XD para que o IMS possa usar o índice em resposta a uma solicitação DL/I.
- Dada a existência de um índice secundário, o usuário pode escolher o processamento do banco de dados correspondente na sequência secundária definida por aquele índice. Se o segmento alvo do índice não for a raiz física, a seleção da sequência secundária irá causar uma reestruturação da hierarquia, como explicado na Seção 21.4; em particular, o segmento alvo do índice tornar-se-á a raiz desta reestruturação. Na sequência secundária, o campo indexado atua como o campo de ordenação do segmento raiz; o usuário vê exatamente tantos registros no banco de dados quantas são as ocorrências do campo indexado no banco de dados físico, e esses registros são ordenados pelos valores ascendentes daquele campo. Os valores do campo indexado são retornados como parcela da raiz da chave totalmente concatenada na área de recebimento da chave. A sequência secundária e a estrutura secundária (se aplicável) são especificadas no PCB; observe que aqui temos uma situação em que a visão definida no PCB não é apenas um simples subconjunto da visão definida no DBD básico.
- Relembro ao leitor o AND independente, que pode ser útil se o campo XD corresponder a um campo de um dependente físico do segmento alvo.

Concluindo, vamos tentar relacionar os conceitos de indexação secundária implementados no IMS com a arquitetura ANSI/SPARC apresentada no Capítulo 1. No nível *externo* o usuário não mais fica restrito à sequência primária de processamento, nem à estrutura hierárquica primária (onde primária refere-se ao que está definido no nível *conceptual*). Entretanto, sequência secundária e estrutura secundária só estão suportadas no nível externo — isto é, pode ser definido um esquema externo apropriado e feito um mapeamento ao esquema conceitual — se existir um índice apropriado, sendo que o índice tem que ser considerado como parte do nível conceitual, e não apenas do nível de armazenamento (interno). Mais explicitamente, o usuário no nível externo tem que saber da exis-

tência do índice secundário — o uso do índice não é automático, tendo que ser *forçado* pela referência ao campo XD. Em outras palavras, a decisão de usar ou não o índice fica nas mãos do programador de aplicações, ao invés de estar sob controle do sistema. Isto não é bom, pois não somente os programas que usam um índice secundário perdem algo em termos de independência de dados, como também, como já vimos, o desempenho do sistema pode ser criticamente dependente de uma escolha criteriosa de quando usar ou não um índice em particular.

EXERCÍCIOS

Os exercícios a seguir estão baseados no banco de dados de publicações (veja o Exercício 16.2). Os nomes de segmentos e campos são os definidos na resposta do Exercício 16.2.

21.1 Deverá ser montado um índice secundário para o banco de dados. Que estrutura o usuário verá se:

- O segmento SUB for indexado no campo AUTHNAME (no segmento AUTHOR);
- O segmento PUB for indexado nesse campo;
- O segmento AUTHOR for indexado nesse campo?

21.2 Para o caso (b) acima, mostre qual o DBD para o índice, e os acréscimos necessários no PUB-DBD (o DBD do banco de dados de publicações). Pode ser considerado que o banco de dados esteja em HDAM ou HIDAM, não em HISAM. Mostre também um PCB correspondente.

21.3 Usando a estrutura secundária do caso (b) acima, obtenha todos os nomes de publicações de um determinado autor (digamos Adams).

21.4 Para a mesma estrutura, mostre a forma da chave totalmente concatenada retornada na recuperação de cada um dos tipos de segmentos envolvidos.

21.5 Que restrições se aplicam ao uso de operadores DL/I sobre esta estrutura?

21.6 Se o banco de dados de publicações contiver:

100 segmentos SUB,
uma média de 100 segmentos PUB por SUB,
uma média de 1,5 segmentos DETAILS por PUB,
uma média de 1,2 segmentos AUTHOR por PUB,
quantos segmentos de cada tipo serão vistos na estrutura secundária do caso (b) no Exercício 21.1?

21.7 Com base no Exercício 21.3 acima, compare e contraste o uso de um banco de dados lógico (veja o Exercício 20.1) para obter o mesmo resultado.

REFERÊNCIAS E BIBLIOGRAFIA

Veja [19.1].

22

Bancos de Dados IMS Fast Path

22.1 O DISPOSITIVO FAST PATH

Fast Path é um dispositivo opcional do IMS especialmente projetado para alguns tipos de sistemas *on-line*. Por “sistemas *on-line*” queremos significar sistemas que são conduzidos pelos usuários finais – ou seja, que operam em reação a comandos introduzidos pelos usuários finais a partir de terminais *on-line*. Como um exemplo, consideremos um sistema bancário, no qual os usuários finais são funcionários do banco que podem emitir comandos, por exemplo, de “apresente o saldo da conta A”, “deposite x cruzeiros na conta A”, “retire x cruzeiros da conta A”, e assim por diante. Usualmente o programa de aplicação que manuseia esses comandos é de estrutura bastante simples; basicamente tudo o que tem que fazer é interpretar a mensagem (comando) do terminal, executar as operações desejadas sobre o banco de dados, enviar uma resposta de volta ao terminal de origem e então esperar pela próxima mensagem para reiniciar o ciclo. O processamento efetuado em função de uma mensagem de entrada é chamado de uma *transação*.¹ Observe, incidentalmente, que a porção da transação relativa ao banco de dados é freqüentemente muito simples, algumas vezes envolvendo nada mais do que uma única leitura ou uma única atualização.

Embora as transações sejam individualmente bastante simples, o sistema pode ser solicitado a manusear uma *razão* de transações muito alta (digamos milhares de transações por minuto). Novamente o ambiente bancário proporciona um bom exemplo, pois várias centenas de terminais nas filiais de um banco podem se comunicar cada um centenas de vezes durante o dia com o sistema central, iniciando uma transação a cada vez. O dispositivo Fast Path objetiva justamente aplicações que tenham esta combinação de exi-

¹ A documentação do IMS geralmente usa “transação” para se referir à própria mensagem de entrada, ao invés de ao processamento provocado por ela. Outros sistemas – por exemplo o Sistema R – usam o termo no nosso sentido.

gências (elevada razão de transações e processamento relativamente simples do banco de dados).²

O dispositivo Fast Path oferece tanto possibilidades especiais para comunicação de dados (fora do escopo deste livro) quanto duas estruturas especiais de bancos de dados, o banco de dados da Memória Principal (MSDB) e o banco de dados de Entrada de Dados (DEDB). Os bancos de dados da Memória Principal e os de Entrada de Dados possuem uma estrutura mais simples do que os outros bancos de dados IMS, e oferecem melhor desempenho para certos tipos de operações. Resumidamente, um banco de dados da Memória Principal é um banco de dados apenas de raízes, mantido na memória principal (virtual) durante a operação do sistema; e o banco de dados de Entrada de Dados é uma forma restrita de hierarquia mantida na memória secundária da forma usual, mas particionado em áreas para aumento de disponibilidade e outras razões. As Seções 22.2 e 22.3 discutirão com mais detalhes estes novos tipos de bancos de dados.

22.2 BANCOS DE DADOS NA MEMÓRIA PRINCIPAL

Como mencionamos na seção anterior, um banco de dados da memória principal (MSDB) é um banco de dados apenas de raízes mantido na memória principal durante a execução do sistema (assim reduzindo a atividade de I/O e os tempos de acesso). São boas candidatas para implementação como MSDB pequenas tabelas contendo informações de referência — por exemplo, horários, tabelas de conversão de moedas, tabelas de juros. Opcionalmente é possível estabelecer-se um relacionamento de um-para-um (para um determinado MSDB) entre as ocorrências de segmentos daquele MSDB e os terminais do sistema, caso em que uma determinada ocorrência de segmento só pode ser atualizada por transações oriundas de mensagens de entrada do terminal correspondente. Este arranjo é adequado quando cada terminal requer sua própria área de memória dedicada; um exemplo é o sistema de caixa pagador, no qual tem que ser mantido um registro para cada terminal sobre o numerário disponível naquele terminal. Entretanto, vamos a seguir restringir nossa atenção aos casos em que não existe esse relacionamento.

A Fig. 22.1 mostra um exemplo de DBD para um MSDB de “conta bancária”. Observe:

1. A entrada ACCESS (MSDB) na instrução DBD;
2. A entrada REL = NO na instrução DATASET, que indica não haver relacionamento específico segmento-terminal neste MSDB;
3. A entrada TYPE (F) para o campo BALANCE. Os campos em um MSDB podem ter os tipos F (uma palavra em ponto fixo binário) ou H (meia palavra em ponto fixo binário), além dos tipos usuais do IMS — C, X e P. Além disso, as comparações de SSAs envolvendo um campo numérico (tipos F, H e P) são comparações aritméticas verdadeiras, e não as comparações usuais do IMS de sequência de bits da esquerda para a direita.

² Não se infira desta afirmativa que aplicações mais convencionais não possam usar o dispositivo Fast Path. Pelo contrário, é comum encontrar-se bancos de dados Fast Path sendo usados por aplicações batch comparativamente longas, bem como por aplicações curtas, de transações simples, do tipo que apresentamos (ou até no lugar destas). Por exemplo, o sistema bancário mencionado poderia correr transações simples (tais como “depósito”, “retirada” etc.) durante o expediente diurno, e aplicações batch longas (tal como “impressão de extratos”) durante a noite.

```

DBD      NAME=ACCOUNTS, ACCESS=MSDB
DATASET  REL=NO
SEGMENT  NAME=ACCOUNT, BYTES=56
FIELD    NAME=(ACCCNO, SEQ), BYTES=6, START=1
FIELD    NAME=NADDR, BYTES=46, START=7
FIELD    NAME=BALANCE, BYTES=4, START=53, TYPE=F

```

Fig. 22.1 DBD do MSDB ACCOUNTS.

Os PCBs para um MSDB seguem essencialmente as mesmas regras que os PCBs para qualquer outro tipo de banco de dados, com a exceção de que as únicas PROCOPTS válidas são G (get) e R (replace). R tem que ser especificado se for usada a operação FLD/CHANGE (veja abaixo). Não são permitidas para o MSDB as operações DL/I ISRT e DLET se REL = NO. (Quando não está na memória principal, um MSDB reside em um arquivo sequencial comum. Este arquivo é criado por um utilitário especial, e *não* por operações DL/I ISRT.)

Vejamos agora as operações DL/I que podem ser usadas com um MSDB. Já mencionamos que ISRT e DLET não se aplicam. GNP também é obviamente inválido. As restantes operações “get” (GU, GN, GHU, e GHN) e “replace” (REPL) são permitidas – embora as SSAs sejam restritas ao máximo de uma comparação (sem ANDs nem ORs) e os códigos de comando não poderem ser usados. Além disso, as operações “get hold” e REPL devem ser evitadas, se possível, pois elas causam atrasos nas transações que correm concorrentemente, reduzindo assim o volume global de trabalho do sistema. Para explicar esta observação, temos que introduzir o conceito de *ponto de sincronização* (usualmente abreviado como “synchpoint”).

Genericamente falando, um ponto de sincronização corresponde ao *final de transação*.³ Ele é indicado ao IMS ou por uma operação especial DL/I SYNC, ou por uma operação DL/I, não discutida neste livro, que solicita a próxima mensagem de entrada do terminal. (Qual destas duas possibilidades se aplica a uma determinada situação depende de um parâmetro especificado externamente, que não discutiremos.)

Quando uma transação T1 executa com sucesso um “get hold” em um dado segmento, este torna-se *locked* (bloqueado). O bloqueio é um mecanismo para proteger as transações de interferências por parte de outras transações executando concorrentemente – isto é, a presença de uma transação no sistema não deve fazer com que alguma outra transação produza resultados incorretos. A finalidade do bloqueio no exemplo é basicamente a de garantir que nenhuma outra transação T2 possa atualizar o segmento (em particular, para garantir que, se T1 atualizar o segmento, nenhuma outra transação T2 possa destruir aquela atualização gravando sobre ela). A regra é que uma transação tem que manter um bloqueio sobre um segmento, para poder atualizá-lo. Se T1 atualizar o segmento, via REPL, as transações concorrentes não poderão sequer *ler* aquele segmento, sem falar em atualizá-lo, até que seja desfeito o bloqueio. Para segmentos MSDB, os bloqueios não são desfeitos até que a transação que o mantém atinja o seu próximo ponto de sincro-

³ O conceito de ponto de sincronização aplica-se a qualquer transação, não somente às que operam sob Fast Path em um MSDB.

nização.⁴ Qualquer transação concorrente que busque acesso a tal segmento terá simplesmente que esperar até que ocorra o ponto de sincronização. São essas esperas que provoca(m) uma redução no volume de trabalho anteriormente mencionado.

A operação FLD objetiva evitar os atrasos causados por essas esperas. Resumidamente, a FLD permite que uma transação *solicite* uma atualização sobre um segmento MSDB, mas não executa essa atualização até que a transação atinja o seu próximo ponto de sincronização,⁵ e não bloqueia o segmento. (O problema da interferência entre transações é manuseado de forma diferente, como veremos.) Assim, as transações concorrentes não ficam bloqueadas em relação ao segmento, não ocorrendo portanto os atrasos causados pelo bloqueio.

Uma operação FLD consiste logicamente de duas ou mais suboperações, uma para selecionar (mas não recuperar) o segmento desejado, e uma ou mais suboperações CHANGE e/ou VERIFY sobre o segmento. Por exemplo, a operação FLD

```
FLD      ACCOUNT (ACCNO='729835'):  
          CHANGE (BALANCE=0)
```

que muda para zero o saldo (balance) no segmento da conta 729835, pode ser vista como uma suboperação⁶ “get unique” seguida por uma suboperação “replace” – exceto que (a) o segmento não é recuperado como o seria em uma GU genuína, e (b) a modificação não ocorre até o próximo ponto de sincronização. (A expressão “CHANGE (BALANCE = 0)” é um exemplo de um *argumento de pesquisa de campo* (FSA). Como usualmente fazemos, usamos uma versão simplificada da sintaxe real do DL/I.)

A execução de uma operação FLD faz com que sejam retornados à transação diversos códigos de status – um código para cada suboperação CHANGE ou VERIFY distinta, além de um código resumo global. O código resumo é retornado ao PCB da forma usual. Os outros códigos são retornados em espaços especiais dentro dos argumentos de pesquisa de campos; para detalhes, veja [22.1].

Em geral, uma operação FLD/CHANGE pode atualizar um campo especificado ajustando-o para um valor específico, como acima, ou por aumento ou redução de um determinado valor. Os três casos são representados, respectivamente, pelos operadores =, +, e -. Por exemplo, para subtrair \$ 60 do saldo da conta 729835:

```
FLD      ACCOUNT (ACCNO='729835'):  
          CHANGE (BALANCE-60)
```

⁴ Em outros tipos de bancos de dados, o bloqueio é desfeito antes do próximo ponto de sincronização, desde que o segmento não tenha sido atualizado.

⁵ Em um MSDB, a operação REPL também não atualiza o segmento até o próximo ponto de sincronização. Uma razão para a postergação da atualização é que o IMS pode decidir durante o processamento do ponto de sincronização que toda a transação tenha que ser refeita desde o início (veja mais adiante nesta seção). Portanto, a semântica da seqüência GHU-REPL-GU, em que as três operações se referem ao mesmo segmento, depende de ser o banco de dados envolvido um MSDB ou um dos outros tipos (no MSDB, o GU verá o segmento exatamente como ele era antes da emissão do REPL).

⁶ Se a SSA especificar uma condição de igualdade na chave (que tem que ser única, incidentalmente; chaves não-únicas não são permitidas em um MSDB), o IMS usará uma pesquisa binária para localizar o segmento desejado.

FLD/VERIFY é usado para verificar se o segmento satisfaz a alguma condição específica. Por exemplo:

```
FLD ACCOUNT (ACCNO='729835'):  
    VERIFY (BALANCE>60)
```

(Naturalmente, FLD/VERIFY tem que ser seguido por um teste do código de retorno apropriado.) É claro que um VERIFY da forma mostrada teria que ter precedido o CHANGE de "redução de saldo" mostrado anteriormente. Na prática, as duas etapas seriam normalmente combinadas em uma única operação FLD:

```
FLD ACCOUNT (ACCNO='729835'):  
    VERIFY (BALANCE>60),  
    CHANGE (BALANCE-60)
```

Uma única operação FLD pode incluir qualquer quantidade de VERIFYs e/ou CHANGES. Os VERIFYs são executados primeiro. Se qualquer dos VERIFY falhar, não ocorrerá a execução de nenhum dos CHANGES daquela operação FLD. Se todos os VERIFY forem bem-sucedidos, os CHANGES serão executados na seqüência em que foram escritos.

Consideremos agora a seguinte situação. Suponhamos que o saldo atual da conta 729835 seja de \$100, e suponhamos que duas transações distintas A e B são iniciadas (a partir de dois terminais diferentes) aproximadamente ao mesmo tempo, ambos tentando retirar \$60 desta mesma conta. Serão iniciadas duas cópias da aplicação de retiradas, que correrão concorrentemente para estas duas transações. Suponhamos que a seqüência de eventos seja a mostrada na Fig. 22.2.

A Fig. 22.2 deve ser interpretada como se segue.

- Transação A: saldo > 60? (Sim)
- Transação A: subtraia 60 do saldo. (OK, mas a atualização não é realizada neste momento)
- Transação B: saldo > 60? (Sim, pois ainda vale 100, apesar da etapa anterior)
- Transação B: subtraia 60 do saldo. (OK, mas a atualização não é realizada neste momento)
- A transação A se completou
- A transação B se completou
- Começa o processamento do ponto de sincronização da transação A. Durante esse processamento, são bloqueados todos os segmentos a serem atualizados, sendo repetidos todos os VERIFYs da transação – neste caso, com sucesso. São então aplicados os CHANGES da transação (fazendo com que o saldo da conta torne-se agora 40), liberados os bloqueios, e terminado o processamento do ponto de sincronização. *Observe que o processamento para o ponto de sincronização de B não pode ocorrer até que termine o processamento para o ponto de sincronização de A;* se B tentar processar seu ponto de sincronização antes que A tenha terminado, como na Fig. 22.2, então B é colocado em estado de espera (porque A está mantendo um

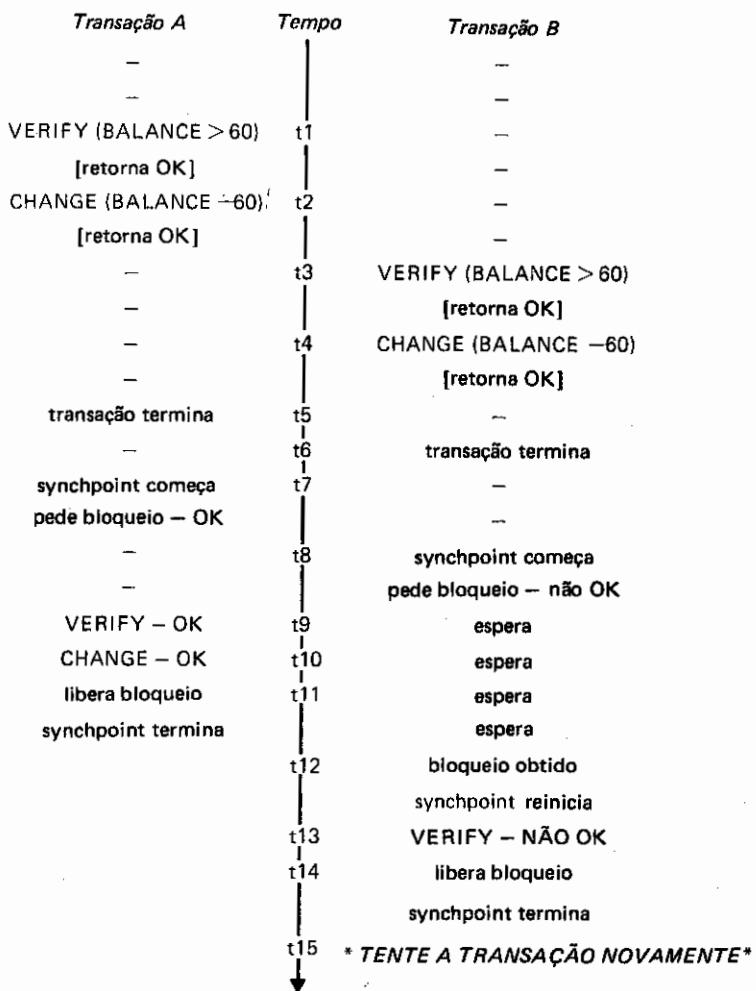


Fig. 22.2 Exemplo da necessidade de ser refeita uma transação.

bloqueio sobre o segmento da conta, e B necessita dele). Em outras palavras, embora as transações A e B sejam “concorrentes” (isto é, executam em paralelo), o processamento para os pontos de sincronização dessas duas transações não é efetivamente concorrente, mas sim executado na base de uma transação por vez.

Começa o processamento do ponto de sincronização da transação B. O segmento da conta é bloqueado. Se qualquer dos VERIFYs falhar — o que é o caso neste exemplo — então o sistema “desfaz” a transação, e volta a tentar executá-la. O processo de desfazer consiste da destruição de todas as atualizações CHANGE e REPL pos-

tergadas,⁷ destruição de qualquer mensagem que a transação tenha gerado para transmitir de volta ao terminal, e reexecução do programa desde o início, entregando-lhe a mensagem original de entrada para novo processamento. No exemplo, esta segunda execução receberá um código de falha do VERIFY (porque o saldo agora é de 40). O programa poderá agora tomar o caminho apropriado — por exemplo, mandar de volta ao terminal uma mensagem de “INSUFICIÊNCIA DE FUNDOS”. (Caso o programa já tenha executado ou solicitado alguma atualização no banco de dados [sobre qualquer banco de dados, MSDB ou outro] ele poderá ter que cancelar essas atualizações neste ponto. A instrução DL/I especial ROLB é fornecida para permitir que o programa desfaça as atualizações. Observe que é da responsabilidade do programa de aplicação emitir o ROLB se necessário — o sistema *não* cancelará automaticamente as atualizações durante o processamento do ponto de sincronização, pois este tem que supor, por definição, que todas as solicitações de atualizações de transação realmente significam o que indicam.)

Das explicações anteriores podemos ver que, do ponto de vista de uma transação determinada, o fato de ser satisfeita uma certa condição (por exemplo, $BALANCE > 60$) no MSDB em um determinado momento t *não* garante que a mesma condição seja ainda satisfeita em um momento posterior t' . Se a lógica da transação exigir essa garantia, pode ser usado “get hold” ao invés de VERIFY. “Get hold” garante que nenhuma transação concorrente poderá alterar o segmento envolvido antes que esta transação se complete. No entanto, como mencionado anteriormente, “get hold” deve ser usado o menos possível em um MSDB.

22.3 BANCOS DE DADOS DATA ENTRY

Um banco de dados de Entrada de Dados (DEDB) é semelhante em alguns aspectos a um banco de dados HDAM — mas um banco de dados HDAM que é tanto estendido quanto restrito de várias formas. As principais extensões são: (a) O DEDB fornece suporte especial a segmentos “dependentes seqüenciais”; (b) O DEDB pode ser particionado em até 240 *áreas*. A principal restrição é que a estrutura hierárquica do banco de dados fica limitada a um tipo de segmento raiz (em que é exigido chave única), com zero a sete tipos de segmentos filhos imediatos. (Portanto a hierarquia possui no máximo dois níveis.) Desses tipos de segmentos filhos, pelo menos um, o mais à esquerda, é um tipo de segmento “dependente seqüencial”; os demais, que se comportam essencialmente como segmentos de um HDAM comum, são tipos de segmentos “dependentes diretos”.

Os segmentos dependentes seqüenciais destinam-se a suportar aplicações de “captura de dados”, nas quais grandes volumes de dados são entrados por terminais para posterior processamento em *batch*. Podem também ser úteis para preparar um arquivo de auditoria, particularmente por não poderem ser atualizados (veja adiante). Tipicamente, cada terminal corresponde a um segmento raiz particular no banco de dados, e todos os dados entrados por aquele terminal são armazenados como dependentes seqüenciais daquela raiz. Os dependentes seqüenciais são encadeados a partir de sua raiz em seqüência de último-a-chegar/primeiro-a-sair (veja abaixo); portanto a inserção de um novo segmento é

⁷ Desfazer uma transação que já tenha atualizado um banco de dados não MSDB é mais complicado; como essas atualizações não são postergadas mas sim aplicadas no momento da solicitação, elas têm que ser desmanchadas, e não apenas “esquecidas”.

rápida. É fornecido um utilitário especial, que executa a recuperação massiva dos dependentes seqüenciais, copiando-os para um arquivo seqüencial para posterior processamento em *batch*.

O particionamento em áreas oferece uma série de vantagens operacionais. Cada área é um arquivo VSAM separado, e cada registro do banco de dados (raiz mais seus dependentes) fica totalmente contido dentro de uma área. A área correspondente a um determinado registro é determinada por uma rotina de randomização fornecida pelo DBA; assim, por exemplo, uma área poderia conter registros de clientes pessoas jurídicas, e outra registros de clientes pessoas físicas. O particionamento não é visível ao programador de aplicações. Daremos alguns exemplos das vantagens que o conceito de área oferece.

- Podem ser designadas áreas diferentes para tipos de dispositivos diferentes (no exemplo de clientes acima, se os clientes pessoas jurídicas representarem 90 por cento de todo o processamento, poderá ser vantajoso manter esses registros em um dispositivo mais rápido).
- Os parâmetros de alocação de espaço podem variar de área para área (por exemplo, algumas áreas podem ter proporcionalmente mais espaço para dependentes seqüenciais do que outras).
- As operações de reorganização e recuperação podem ser executadas sobre áreas individuais, ou sobre *subconjuntos* de áreas individuais, ao invés de sobre todo o banco de dados; dessa forma não é necessário que todo o banco de dados fique indisponível quando são executadas essas operações.
- Podem ser suportados bancos de dados extensos (maiores do que a capacidade de um único arquivo VSAM).
- Nem todas as áreas precisam estar *on-line* simultaneamente.

Como já mencionado, o acesso às raízes de um DEBD é fornecido por uma rotina de randomização fornecida pelo DBA, como em HDAM. O acesso a partir de uma dada raiz aos seus dependentes é fornecido por indicadores de localização filho/gêmeo. Os dependentes seqüenciais de uma dada raiz são encadeados em seqüência último-a-chegar/primeiro-a-sair (isto é, a inserção mais recente aparece na frente da cadeia, próxima à raiz); não pode ser definido campo de ordenação para esses segmentos. Os dependentes diretos (de um determinado tipo) terão ou um campo de ordenação único, caso em que o IMS mantém a seqüência gêmea da forma usual, ou não terão qualquer campo de ordenação, caso em que a seqüência gêmea tem que ser mantida por programa (veja detalhes em [22.1]).

Cada área do DEBD é dividida em três partes: uma *parte de raiz endereçável*, uma *parte independente de excedentes*, e uma *parte de seqüenciais dependentes*. As duas primeiras são análogas à área endereçável do segmento raiz e à área de excedentes em HDAM (veja o Capítulo 19), exceto que em um DEBD a rotina de randomização tem que gerar um número de área, além do endereço do registros dentro daquela área. A parte de seqüenciais dependentes guarda todos os segmentos dependentes seqüenciais de todas as raízes da área, encadeados às suas raízes como explicado anteriormente, mas armazenados na seqüência de momento-da-chegada (portanto segmentos fisicamente adjacentes podem pertencer a raízes diferentes).

Vamos considerar brevemente o DBD para um DEBD. Primeiro, a instrução DBD em si tem que especificar ACCESS = DEBD e uma rotina de randomização (via entrada

RMNAME, como em HDAM). Segundo, a instrução DBD tem que estar seguida por 1 a 240 instruções AREA, cada uma contendo uma entrada DD1 (semelhante à entrada DATASET DD1 – veja o Capítulo 19) e vários detalhes da área em si, tais como tamanho de suas partes componentes. Terceiro, as instruções SEGMENT e FIELD são basicamente as de sempre, exceto que a instrução SEGMENT para o dependente seqüencial, se houver, tem que incluir a entrada TYPE = SEQ.

Quanto ao PCB, observemos que as únicas PROCOPTS válidas para um segmento dependente seqüencial são G (get) e I (insert). Não podem ser usados nesses segmentos REPL e DLET. (A remoção de uma raiz em um DEDB não causa necessariamente a remoção de todos os dependentes seqüenciais daquela raiz. Veja maiores explicações em [22.1].) A PROCOPT P tem uma interpretação especial para um DEDB; novamente veja detalhes em [22.1].

Finalmente, consideremos as operações DL/I. Afora as restrições sobre segmentos dependentes seqüenciais feitas acima, são válidas todas as operações DL/I usuais – exceção que, como nos bancos de dados da Memória Principal, as SSAs ficam restritas ao máximo de uma comparação (sem ANDs ou ORs), e os códigos de comandos não podem ser usados. Entretanto, operações de recuperação seqüencial (GN ou GNP) sobre os dependentes seqüenciais devem ser evitadas se possível, pois elas tendem a ser muito lentas (como consequência da estrutura de armazenamento que, como já explicado, foi especificamente projetada para rápida inserção). Uma operação especial, POS, é fornecida para permitir que o usuário descubra quanto espaço já foi utilizado dentro da parte de dependentes seqüenciais de uma determinada área.

REFERÊNCIAS E BIBLIOGRAFIA

Veja também [16.1], [18.1], e [19.1].

22.1 IBM Corporation. IMS Version 1 Release 1.5 Fast Path Feature Description and Design Guide. IBM Form nº G320-5775.

Parte 4

A Abordagem de rede

A maior parte do interesse corrente sobre a abordagem de rede originou-se na publicação do DBTG Report [23.1] em abril de 1971. Na Parte 4 veremos as especificações DBTG detalhadamente. Por isso, como na Parte 3 muitos detalhes apresentados serão específicos do sistema, mas os conceitos básicos podem ser considerados como típicos de qualquer sistema de rede. De qualquer forma, o DBTG é sem dúvida o exemplo mais importante desta abordagem. O capítulo 23 descreve a estrutura global de um sistema DBTG; os Capítulos 24 e 25 discutem a estrutura de dados DBTG (nos níveis "conceitual" e externo); e o Capítulo 26 introduz a linguagem de manipulação de dados.

Vários outros sistemas de redes estão descritos nas referências do Capítulo 1.

23

A Arquitetura de um Sistema DBTG

23.1 HISTÓRICO

A sigla DBTG vem do “Data Base Task Group” do CODASYL COBOL Committee (CC). O CC é o grupo responsável pelo desenvolvimento da linguagem COBOL; suas atividades são documentadas no *COBOL Journal of Development*, publicado a cada dois ou três anos, que serve como a especificação oficial da linguagem COBOL. O Data Base Task Group, embora sendo um grupo de trabalho do CC, não restringe sua atenção somente ao COBOL. De fato, o relatório final do DBTG [23.1], que surgiu em abril de 1971, continha propostas para três linguagens distintas: a linguagem de descrição de dados esquema (schema DDL), uma linguagem de descrição de dados subesquema (subschema DDL), e uma linguagem de manipulação de dados (DML). Destas, a segunda e terceira consistem basicamente de extensões ao COBOL, mas a primeira é definitivamente uma linguagem distinta e autocontida, embora guardando um certo sabor de COBOL.

As finalidades dessas linguagens encontram-se a seguir. A schema DDL é uma linguagem para descrição de um banco de dados estruturado em rede; o “esquema” DBTG é portanto análogo ao esquema conceitual ANSI/SPARC. A schema DDL objetiva atingir as exigências de diversas linguagens de programação distintas, não somente o COBOL – como no IMS, o “usuário” em um sistema DBTG é um programador de aplicações comum – e por isso a linguagem não está polarizada no sentido de qualquer linguagem de programação específica. A subschema DDL, por outro lado, é uma linguagem para definição de visões externas (o “subesquema” DBTG corresponde ao esquema externo ANSI/SPARC), e portanto tem que ter uma sintaxe compatível com a de alguma linguagem específica de programação, como explicamos no Capítulo 1. De forma semelhante, a linguagem de manipulação de dados tem que ter uma sintaxe compatível com a de alguma linguagem principal. A DML e a DDL subesquema definidas em [23.1] tiveram como objetivo seu uso com COBOL; o Task Group esperava que outros organismos definissem DMLs e DDLs subesquema para uso com outras linguagens de programação, como o PL/I.

Próximo ao final de 1971, foi estabelecido um novo comitê CODASYL, o Data Description Language Committee (DDLC). O DDLC foi formado para executar, em relação

à DDL esquema, o que o CC faz com relação ao COBOL; em outras palavras, ele é responsável pela continuidade de desenvolvimento da DDL esquema e pela produção correspondente de um *Journal of Development*, documentando o estado atual da linguagem. A primeira versão deste jornal apareceu em 1973.

A DML e DDL subesquema COBOL do DBTG fazem agora parte do *COBOL Journal of Development* corrente (1978) [23.3] (sob o nome de “The COBOL Data Base Facility”). Esta versão do COBOL está sendo considerada pelo American National Standards COBOL Committee (X3J4) como base para o próximo COBOL padrão. De forma semelhante, a DDL esquema documentada no *DDL Journal of Development* de 1978 [23.4] está sendo considerado pelo ANS Data Description Committee (X3H2) como a base para um padrão de descrição de dados. É portanto possível que alguma forma das especificações DBTG torne-se um padrão em futuro próximo. (Continuamos a usar DBTG como um rótulo conveniente, embora todos os detalhes de linguagem dos próximos capítulos estejam baseados nos Journals of Development de 1978 e em documentos correntes de trabalho ANS, mais do que no relatório original. Essas especificações mais recentes diferem substancialmente da proposta original de 1971).

Daqui para frente, usaremos “DDL” para significar a DDL esquema, a menos de qualificação diferente, e COBOL para significar a versão do COBOL documentada em [23.3], que inclui a possibilidade de uso de bancos de dados.

23.2 ARQUITETURA

A arquitetura de um sistema DBTG está ilustrada na Fig. 23.1.

A “visão conceitual” (não é um termo DBTG) é definida pelo *esquema*. O esquema consiste essencialmente de definições dos vários tipos de *registros* do banco de dados, dos *itens de dados* que eles contêm e os *conjuntos* nos quais eles são agrupados. (O conceito de conjunto DBTG está explicado em detalhes no Capítulo 24. De forma genérica, é o meio de representação dos relacionamentos em um sistema DBTG.)

A estrutura de armazenamento (visão interna) do banco de dados é descrita pelo *esquema de armazenamento*, escrito em Data Storage Description Language (DSDL – Linguagem de Descrição do Armazenamento dos Dados). O *DDL Journal of Development* de 1978 inclui um apêndice contendo propostas para uma possível DSDL [24.6].

Uma visão externa (não é um termo DBTG) é definida por um *subesquema*. Um subesquema consiste essencialmente de uma especificação sobre quais tipos de registros esquema interessam ao usuário, que itens de dados esquema o usuário deseja ver nesses registros, e que relacionamentos esquema (conjuntos) interligando esses registros o usuário deseja considerar. Ficam automaticamente excluídos todos os outros tipos de registros, itens de dados e conjuntos. Não é possível, pelo menos na DDL subesquema correntemente especificada, definir-se qualquer estruturação no subesquema — por exemplo, um novo tipo de relacionamento (conjunto) ou um registro que se estenda por dois ou mais registros esquema — que não exista explicitamente no esquema.

Finalmente, como explicamos na Seção 23.1, os usuários são programadores de aplicações, que escrevem em linguagem comum de programação, tal como o COBOL, estendida para incorporar a linguagem de manipulação de dados DBTG. Cada programa de aplicação “invoca” o subesquema correspondente; por exemplo, usando o COBOL Data Base Facility, o programador precisa apenas especificar o nome do subesquema desejado na Data Division do programa. Esta invocação fornece a definição da “user work area” (UWA — área de trabalho do usuário) para aquele programa. A UWA contém uma locali-

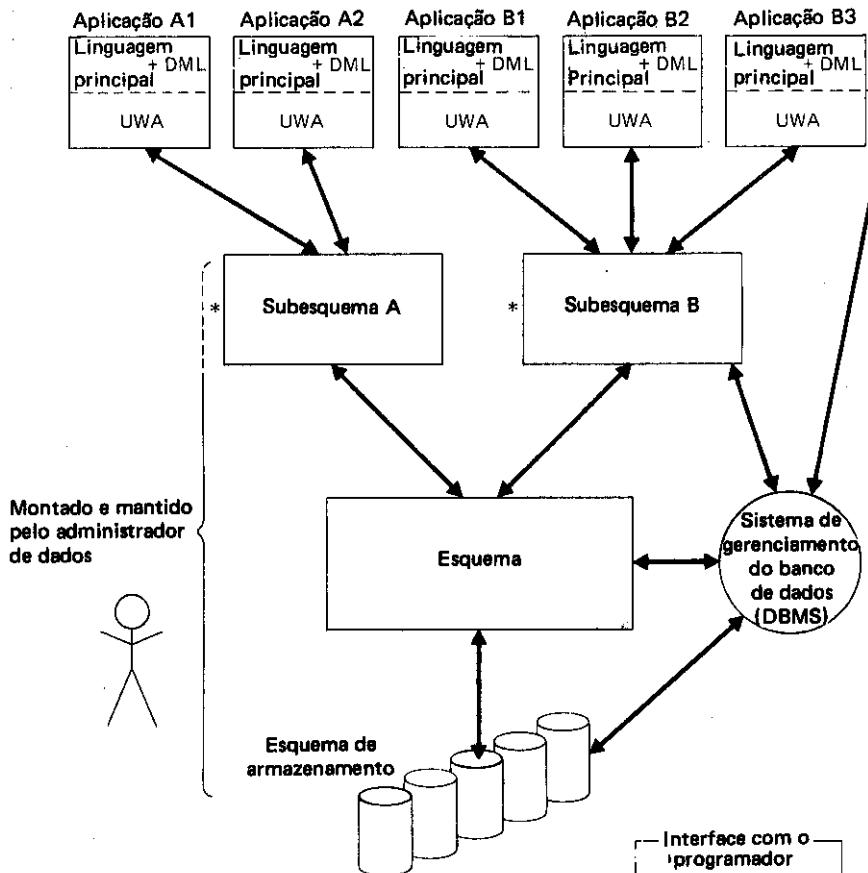


Fig. 23.1 Arquitetura de um sistema DBTG.

zação distinta para cada tipo de registro (e consequentemente para cada tipo de item de dado) definido no subesquema. O programa pode se referenciar a essas localizações de itens de dados e registros por nomes definidos no subesquema. (O termo "user work area" não é usado em COBOL; ao invés, cada tipo de registro possui uma "record area" distinta. No entanto, o conceito é o mesmo.)

Os termos DBTG para "DBA" e "interface com o usuário" são respectivamente "administrador de dados" e "interface com o programador". "DBMS" é, por si mesmo, um termo DDL. O COBOL usa "DBCS" (Data Base Control System) para se referenciar ao componente em tempo de execução do DBMS.

REFERÊNCIAS E BIBLIOGRAFIA

23.1 Data Base Task Group of CODASYL Programming Language Committee. *Report* (abril de 1971). Disponível na ACM, BCS, e IAG. Na época deste relatório, o CODASYL COBOL Committee era conhecido como o Programming Language Committee (PLC).

23.2 R. W. Engles. "An Analysis of the April 1971 DBTG Report – A Position Paper Presented to the Programming Language Committee by the IBM Representative to the Data Base Task Group." *Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and control.*

Engles era o representante da IBM no Data Base Task Group na época em que foi produzido o relatório final [23.1]. Este artigo, denominado artigo da posição IBM, documenta uma série de objeções importantes às propostas de [23.1].

23.3 CODASYL COBOL Committee. *COBOL Journal of Development* (1978).

23.4 CODASYL Data Description Language Committee. *DDL Journal of Development* (1978).

23.5 G. G. Dodd. "APL – A Language for Associative Data Handling in PL/I." *Proc. FJCC*. Montvale, N.J.: AFIPS Press (1966). Linguagem associativa de programação de Dodd, que foi uma das duas maiores influências originais sobre o DBTG (a outra foi IDS [3.3]).

23.6 R. W. Taylor and R. L. Frank. "CODASYL Data Base Management Systems." *ACM Computing Surveys* 8, nº 1 (março de 1976).

23.7 R. W. Engles. "A Description of the COBOL Data Base Facility." *Proc. GUIDE* 47 (novembro de 1978). Disponível na GUIDE International.

23.8 T. W. Olle. *The CODASYL Approach to Data Base Management*. New York: Wiley Interscience (1978).

23.9 CODASYL Fortran Data Base Committee. *Fortran Data Base Facility* (1980).

Proposta para uma DML e DDL subesquema Fortran.

24

Estrutura de Dados DBTG

24.1 INTRODUÇÃO

Este Capítulo está planejado da seguinte maneira: as Seções 24.2, 24.3, e 24.4 estão voltadas para os “conjuntos DBTG”, o dispositivo mais obviamente distinto da estrutura DBTG de dados. Especificamente, a Seção 24.2 mostra como os conjuntos DBTG podem ser usados para se construir estruturas hierárquicas; a Seção 24.3 executa a mesma função para estruturas em rede; e a Seção 24.4 lida com um tipo especial de conjunto, conhecido como conjunto “singular”. A Seção 24.5 apresenta um esquema completo da versão DBTG para o banco de dados de peças e fornecedores discutidos nos capítulos anteriores.

A Seção 24.6 está voltada para o problema de “classe de membro”. Como a classe de membro é especificada no esquema, foi introduzida uma explicação neste Capítulo. Entretanto, o leitor provavelmente terá necessidade de voltar a esta seção após ler, no Capítulo 26, as descrições das instruções DML que são afetadas pela classe de membro. A Seção 24.7 discute SET SELECTION. Novamente pode ser interessante ao leitor rever esta seção depois de passar pelo Capítulo 26, pois SET SELECTION também depende, até certo ponto, de alguns aspectos da DML.

24.2 FORMAÇÃO DO CONJUNTO: EXEMPLOS HIERÁRQUICOS

24.2.1 Hierarquia com um nível dependente — A Fig. 24.1 mostra um banco de dados que contém informações sobre departamentos e empregados.

Este banco de dados contém dois tipos de Registros: DEPT (departamento) e EMP (empregado). Normalmente, cada um destes registros inclui diversos itens de dados,¹ embora no diagrama só esteja mostrado um em cada, ou seja, DNO (nº do departamento) para DEPT, e ENO (número do empregado) para EMP. Há três ocorrências de DEPT e nove de EMP. Estas ocorrências de registros estão grupadas em um conjunto denominado

¹ O DBTG permite que os registros contenham grupos repetidos. Vamos restringir nossa atenção aos registros normalizados nesta parte do livro.

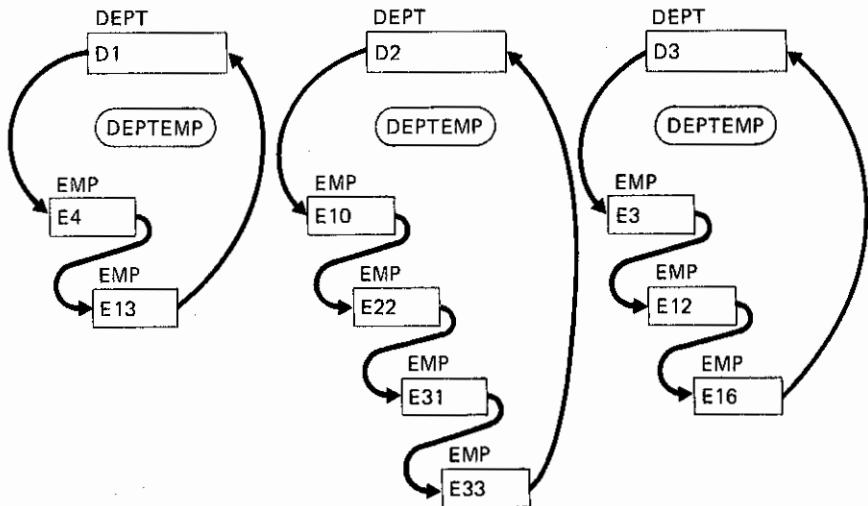


Fig. 24.1 O banco de dados departamentos/empregados.

DEPTEMP. Como já mencionamos, o conjunto é o dispositivo primário de distinção na estrutura DBTG. Cada *tipo* de conjunto é definido no esquema como tendo um determinado tipo de registro que é o *dono*, e um outro tipo de registro que é o *membro*; no exemplo, o tipo de conjunto DEPTEMP seria declarado no esquema com DEPT como seu dono e EMP como seu membro.² Cada *ocorrência* de um determinado tipo de conjunto consiste precisamente de uma ocorrência de seu dono juntamente com zero ou mais ocorrências de seu membro. (O caso zero surgiria se, por exemplo, algum departamento não tivesse empregados. Nessa situação, ainda existiria o conjunto de ocorrências — consistindo somente de uma ocorrência dono — mas seria vazia.) Dentro de um determinado conjunto, nenhuma ocorrência de registro membro pode pertencer a mais de uma ocorrência daquele conjunto em qualquer momento (embora possa pertencer a ocorrências diferentes em momentos diferentes, em geral). Por exemplo, nenhuma ocorrência EMP pode pertencer a mais de uma ocorrência DEPTEMP em um determinado momento.

Cada ocorrência de um conjunto representa um relacionamento hierárquico entre a ocorrência dona e as correspondentes ocorrências membro. No exemplo, naturalmente, o relacionamento é o normal, de departamentos para empregados. O meio pelo qual cada ocorrência dona fica ligada às ocorrências membro correspondentes é irrelevante, no que tange ao usuário. Uma forma de se fazer essas conexões (não a única) é via uma cadeia de indicadores de localização que se origina na ocorrência dona, percorre todas as ocorrências membro, e finalmente retorna à ocorrência dona (como está mostrado na Fig. 24.1). Por simplicidade, este será o método assumido nesta parte do livro; se na prática for adotado algum outro método, ele terá que ser funcionalmente equivalente ao método de ca-

² Estamos ignorando a possibilidade de conjuntos que contenham mais do que um tipo de membro. Veja [23.4] para detalhes sobre tais conjuntos.

deia de indicadores de localização, e por isso esta simplificação é razoável. (O fato é que o usuário sempre poderá considerar como existindo fisicamente as cadeias de indicadores de localização, mesmo que na realidade eles não estejam representados como indicadores de localização na memória. Veja o Capítulo 3.)

O desenho das ocorrências de conjuntos como cadeias de indicadores de localização, como na Fig. 24.1, já se tornou uma convenção aceita. Uma outra convenção amplamente aceita para a representação dos *tipos* de conjuntos é o “diagrama de estrutura de dados” de Bachman [24.1]; como uma ilustração veja a Fig. 24.2, onde a estrutura de DEPTEMP está mostrada como se fosse uma estrutura IMS. As diferenças entre o simbolismo de Bachman e o usado em IMS são que (a) o elo entre o dono e o membro é *rotulado* com o nome do conjunto, enquanto que em IMS esses elos são anônimos, e (b) o elo é *direcionado* para indicar qual é o dono e qual é o membro. (A especificação da direção do elo torna-se necessária em situações mais complexas, onde nem sempre é possível mostrar o dono como estando “acima” do membro.) Fica normalmente mais claro mostrar-se exemplos de ocorrências do conjunto, do que simplesmente mostrar o tipo do conjunto, e geralmente faremos isso em nossos exemplos.

O leitor já deve ter percebido até aqui que o “conjunto” DBTG é bastante diferente do que se entende como conjunto em matemática. A terminologia DBTG é algo infeliz, pois (por exemplo) precisa freqüentemente fazer referência ao conjunto de registros membros — significado normal — pertencente a um conjunto particular — significado DBTG. Para evitar essas confusões, alguns autores usam uma terminologia especial para se referir a construções do DBTG; dentre os termos que já foram usados estão “conjunto DBTG”, “conjunto CODASYL”, “conjunto do banco de dados”, “conjunto da estrutura de dados”, “conjunto acoplado ao dono”, e “conjunto agregado”. Neste livro continuaremos a usar o termo não qualificado “conjunto” em sua maior parte, exceto quando este uso possa provocar ambigüidade.

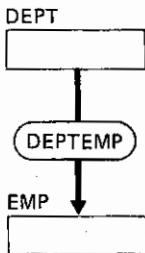


Fig. 24.2 Estrutura do conjunto DEPTEMP.

24.2.2 Hierarquia com mais de um nível dependente — A Fig. 24.3 mostra (parte de) um banco de dados que contém informações sobre divisões, departamentos e empregados.

Este exemplo ilustra o fato de que um tipo particular de registro (DEPT no exemplo) pode ser declarado no esquema como sendo membro de um tipo de conjunto (DIV-DEPT) e dono de outro (DEPTEMP). Duas ocorrências de DIVDEPT e três de DEPTEMP estão mostradas na Fig. 24.3 (*Sempre* é verdade, por definição, que o número de ocorrências de um conjunto é precisamente o mesmo que o número de ocorrências do dono.) Portanto, temos uma estrutura hierárquica com dois níveis dependentes (veja a Fig. 24.4). Geralmente, podemos montar desta forma uma hierarquia com qualquer quantidade de níveis dependentes.

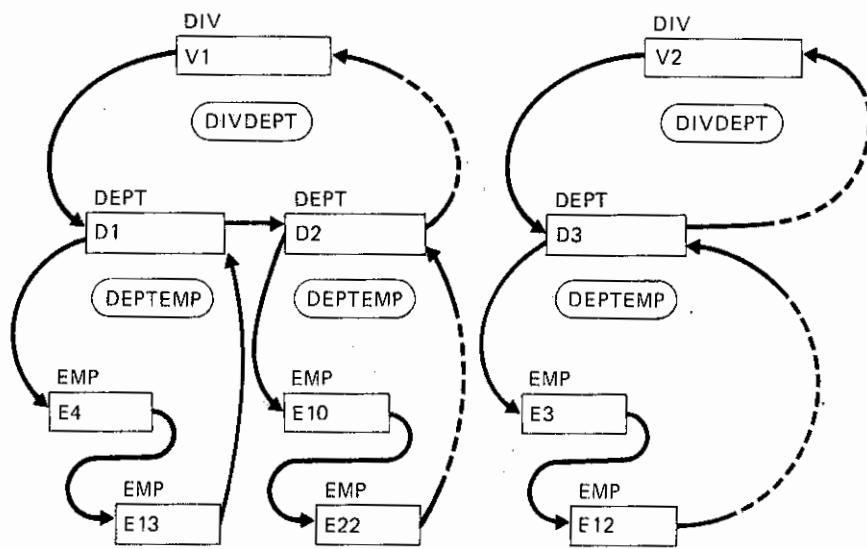


Fig. 24.3 O banco de dados divisão-departamentos-empregados.

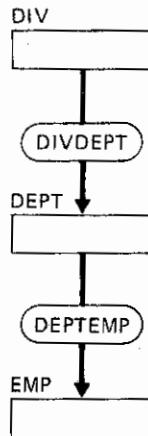


Fig. 24.4 Estrutura dos conjuntos DIVDEPT e DEPTEMP.

24.2.3 Hierarquia com mais de um tipo de registro em um nível dependente — A Fig. 24.5 mostra (parte de) um banco de dados que contém informações sobre empregados, histórico sobre seus trabalhos e histórico sobre sua educação.

Ali, cada ocorrência EMP é dona de dois conjuntos de ocorrências:

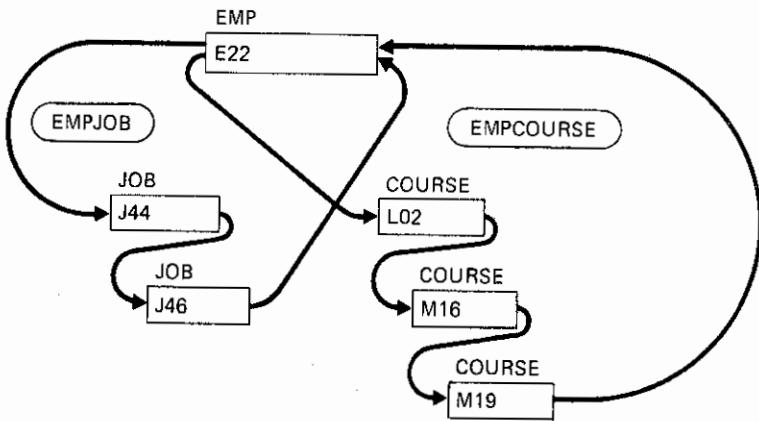


Fig. 24.5 Banco de dados histórico de empregados.

uma ocorrência de EMPJOB, na qual os membros representam os trabalhos realizados pelos empregados, e uma ocorrência EMPCOURSE, na qual os membros representam os cursos feitos pelos empregados. Em geral, determinado tipo de registro (EMP no exemplo) pode ser declarado no esquema como sendo o dono de qualquer quantidade de tipos de conjuntos. Assim, podemos montar estruturas hierárquicas que não somente possuem qualquer quantidade de níveis, mas também qualquer quantidade de tipos de registros em cada nível dependente. Veja a Fig. 24.6.

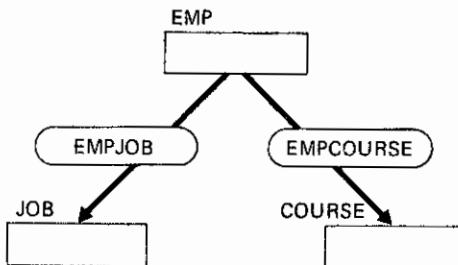


Fig. 24.6 Estrutura dos conjuntos EMPJOB e EMPCOURSE.

Incidentalmente, há uma diferença significativa entre a estrutura DBTG da Fig. 24.6 e a estrutura IMS correspondente. Em DBTG, os dois ramos da hierarquia poderiam ser intercambiados sem causar qualquer efeito à estrutura, enquanto que em IMS esta alteração afetaria a seqüência hierárquica (na qual o usuário vê JOBs precedendo COURSES, por exemplo, se a ramificação JOB encontra-se à esquerda da ramificação COURSE).

24.2.4 Hierarquia com o mesmo tipo de registro em mais de um nível – A Fig. 24.7 mostra (parte de) um banco de dados que contém informações sobre a estrutura gerencial de uma companhia.

Não é permitido que um mesmo tipo de registro seja tanto dono quanto membro de um mesmo tipo de conjunto. Para representar um relacionamento hierárquico entre diferentes ocorrências de um mesmo tipo de registro, é portanto necessário introduzir um nível de via indireta, como ilustra a Fig. 24.7. Ali, o relacionamento a ser representado é o que existe de gerente para empregados (onde o gerente também é considerado como um empregado, e por sua vez tem um gerente, e assim por diante). Introduzimos um segundo tipo de registro (LINK), e definimos dois tipos de conjuntos: EL (dono EMP, membro LINK) e LE (dono LINK, membro EMP). Cada ocorrência de EMP representando um gerente é dona de uma ocorrência EL contendo precisamente um LINK,³, por sua vez, este LINK é dono de uma ocorrência LE cujos membros são as ocorrências EMP representadas no nível inferior.

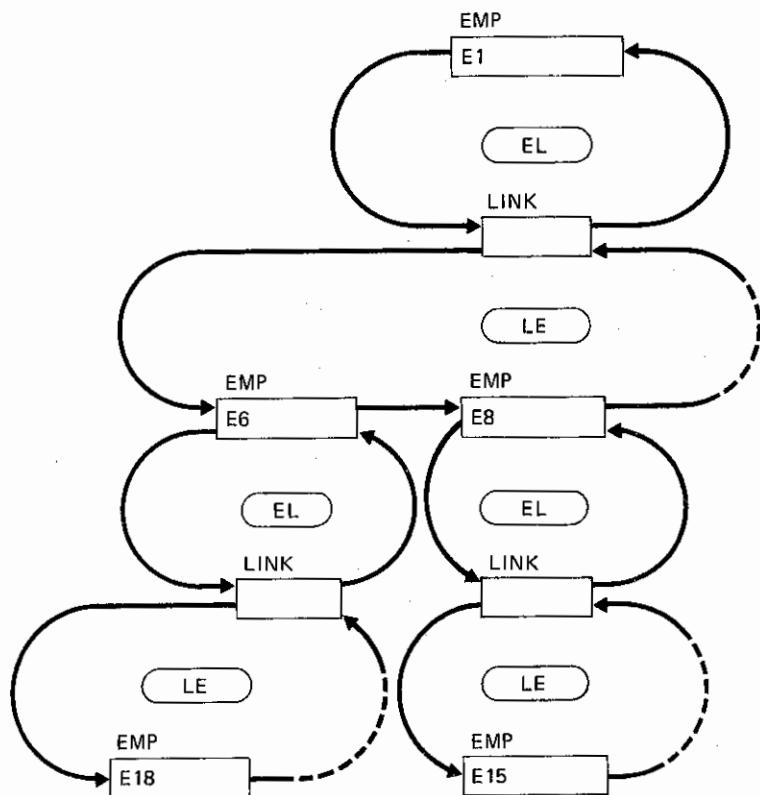


Fig. 24.7 O banco de dados de estrutura gerencial.

³

A manutenção desta correspondência de 1 para 1 entre EMPs e LINKs é da responsabilidade do programa de aplicação, não do DBMS.

tando os subordinados imediatos do gerente. Assim, por exemplo, o empregado E1 é o gerente dos empregados E6, E8, . . . ; E6 é o gerente de E18, . . . ; e assim por diante. A Fig. 24.8 é o diagrama da estrutura de dados. Observe que não existe exigência de que o registro LINK contenha qualquer item de dado.

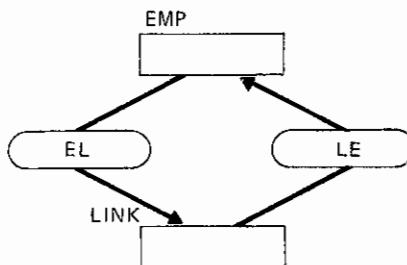


Fig. 24.8 Estrutura dos conjuntos EL e LE.

24.3 FORMAÇÃO DO CONJUNTO: EXEMPLOS EM REDE

24.3.1 Rede envolvendo dois tipos de entidades — A Fig. 24.9 representa um banco de dados que contém os planos correntes para uma série de concertos de orquestra. Temos aqui uma situação de rede típica, pois em geral cada compositor terá trabalhos seus em diversos concertos. Podemos representar esta situação em DBTG introduzindo um tipo de registro de conexão (WORK), cuja função é conectar os dois tipos básicos de entidades (representados pelos tipos de registros CONCERT e COMPOSER, respectivamente). Cada ocorrência de WORK conecta um concerto e um compositor; ela representa a inclusão, no concerto envolvido, do trabalho do compositor envolvido. Introduzimos também dois tipos de conjuntos: CONCW (dono CONCERT, membro WORK) e COMPW (dono COMPOSER, membro WORK).⁴ Para estabelecer uma conexão entre um determinado concerto e um determinado compositor, temos que garantir que a ocorrência WORKpropriada foi entrada na ocorrência do conjunto CONCW para o concerto, e na ocorrência do conjunto COMPW para o compositor. Por exemplo, há uma conexão entre o concerto C1 e o compositor Bartok (veja a Fig. 24.9). Portanto, geralmente as ocorrências do conjunto CONCW de um dado concerto contém ocorrência de WORK para todos os trabalhos naquele concerto, e o conjunto de ocorrências COMPW de um dado compositor contém ocorrências WORK de todas as execuções dos trabalhos daquele compositor em todos os concertos.

A Fig. 24.9 mostra seis ocorrências do conjunto CONCW (dois estão vazios, representando concertos cujos programas ainda não foram planejados) e sete do conjunto COMPW (um está vazio, representando um compositor que se pretende seja incluído em algum dos concertos quando o planejamento estiver completo). Por razões de espaço,

⁴ Não interprete erroneamente da Fig. 24.9 que WORK seja o *dono* de COMPW.

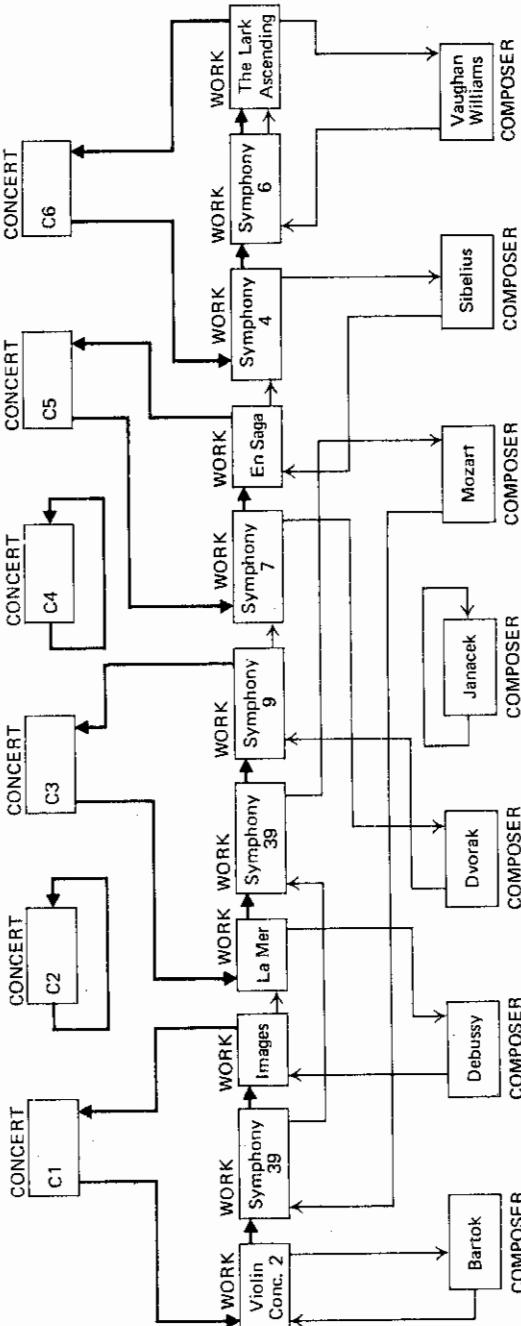


Fig. 24.9 Banco de dados orquestra-concertos.

não estão mostrados os nomes dos conjuntos. Observe que cada ocorrência de WORK contém dados descrevendo a conexão que ela representa, isto é, o nome do trabalho apropriado.

A Fig. 24.10 é o diagrama de estrutura de dados correspondente.

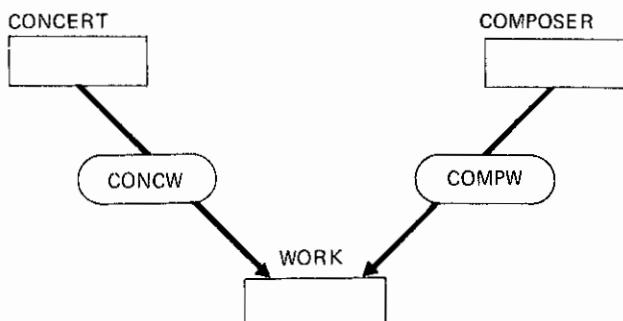


Fig. 24.10 Estrutura dos conjuntos CONCW e COMPW.

24.3.2 Redes envolvendo mais de dois tipos de entidades – O exemplo anterior ilustra o método geral de montagem de um banco de dados em rede em DBTG. Especificamente, se n tipos de entidades (representadas por n tipos de conjuntos) precisam ser conectados, introduzimos um tipo de registro de conexão e n tipos de conjuntos; cada um dos n tipos de registro “entidade” é tornado o dono de um dos tipos de conjuntos, sendo o tipo de registro de conexão feito membro de todos eles; e cada ocorrência de registro de conexão é feita membro de exatamente uma ocorrência de cada um dos n tipos de conjuntos, representando assim a conexão entre as n entidades correspondentes.

O exemplo 24.3.1 pode ser estendido de várias maneiras para ilustrar esta técnica. Seguem-se alguns exemplos em esboço.

- Introdução de um novo tipo de entidade, SOLOIST. Um solista pode tocar em diversos concertos, e alguns concertos podem ter diversos solistas.
- Introdução de um novo tipo de entidade, CONDUCTOR. Cada maestro pode conduzir diversos concertos. (Este relacionamento é em si hierárquico, porém mais do que um maestro pode aparecer em um mesmo concerto, criando uma situação de rede.)
- Introdução de um novo tipo de entidade, ORCHESTRA.
- Introdução de um novo tipo de entidade, WORK-CATEGORY (as possíveis categorias de trabalhos são sinfonia, overture, concerto para violino, e assim por diante). Cada concerto inclui trabalhos de diversas categorias, e cada categoria aparece representada em muitos concertos. Além disso, cada compositor produz trabalhos em diversas categorias, e cada categoria inclui trabalhos de muitos compositores.

Pedimos ao leitor que crie alguns dados de amostragem e monte as ocorrências de conjuntos correspondentes para alguns destes exemplos.

24.3.3 Rede envolvendo somente um tipo de entidade – A Fig. 24.11 representa um banco de dados contendo informações sobre peças e componentes (onde o próprio componente é uma peça e pode ter componentes adicionais, e assim por diante).

Aqui temos uma rede envolvendo somente um tipo de entidade, peças. Entretanto, cada peça está, geralmente, cumprindo dois papéis; ela é a *montagem* de alguns componentes imediatos e também um *componente* de algumas montagens que se seguem. Portanto este é apenas um caso especial da situação de dois tipos de entidades na qual os dois são um e o mesmo, e vamos considerá-lo da mesma forma. Primeiro introduziremos um registro de conexão, LINK. Depois, definiremos dois tipos de conjuntos: BM (lista dos materiais) e WU (onde usado), ambos tendo PART como dono e LINK como membro. Na Fig. 24.11, as ocorrências do conjunto BM estão indicadas pelas setas cheias, e as ocorrências do conjunto WU pelas setas tracejadas. A ocorrência do conjunto BM para uma determinada peça contém uma ligação com cada peça que é componente imediato daquela peça; a ocorrência do conjunto WU para uma determinada peça contém uma ligação com cada peça que contém aquela peça como um componente imediato. Assim, na Fig. 24.11, vemos que P1 contém P2 e P3 como componentes imediatos; inversamente, P3 é um componente imediato de P1 e P2, e P2 por sua vez é também um componente imediato de P1. Os números dentro das ocorrências LINK representam as quantidades correspondentes (por exemplo, cada peça “P1” inclui como componentes duas peças “P2” e quatro peças “P3”).⁵

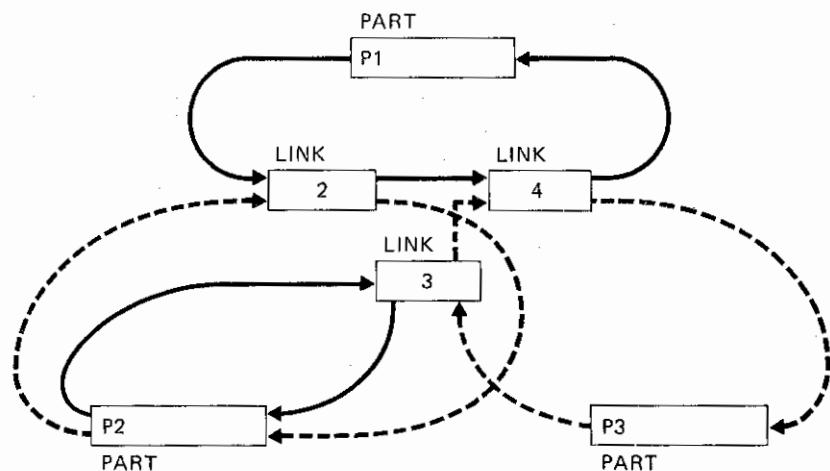


Fig. 24.11 Banco de dados de peças.

5

Lembre-se de que um dado tipo de conjunto não pode ter o mesmo tipo de registro como dono e membro. Por isso uma declaração do conjunto BM (digamos) como tendo dono PART e membro PART não seria uma abordagem aceitável neste exemplo. (Tal abordagem seria inconveniente em qualquer caso, pois não oferece um local óbvio para a informação sobre quantidade.)

O leitor deverá pegar os dados da relação COMPONENT (Fig. 4.4) e montar uma estrutura DBTG equivalente. (Incidentalmente, qual é a representação relacional dos dados da Fig. 24.11?). Observe que temos aqui dois tipos distintos de conjuntos possuindo o mesmo dono e o mesmo membro (naturalmente, os dois tipos de conjuntos representam dois relacionamentos hierárquicos diferentes). O diagrama da estrutura está mostrado na Fig. 24.12.

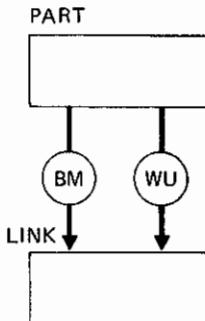


Fig. 24.12 Estrutura dos conjuntos BM e WU.

24.4 CONJUNTOS SINGULARES

A montagem do conjunto introduzida nas duas seções anteriores fornece um meio de se agrupar registros membros de tal forma que os registros em qualquer grupo são logicamente relacionados de alguma forma. Por exemplo, consideremos novamente o caso de departamentos e empregados (Fig. 24.1). Naquele exemplo, cada grupo representa uma coleção de todos os empregados de algum dado departamento. Um operador “encontre o próximo dentro do grupo” na linguagem de manipulação de dados (mais precisamente, uma instrução FIND Formato 3 – veja o Capítulo 26) permitirá que o grupo seja usado como um caminho de acesso aos registros correspondentes; na Fig. 24.1, por exemplo, cada ocorrência do conjunto DEPTEMP fornece um caminho de acesso da ocorrência DEPT às ocorrências EMP relacionadas, e os programas dos usuários podem usar esses caminhos de acesso.

No entanto, na forma como se encontra a Fig. 24.1, não há caminho de acesso conectando *todas* as ocorrências de registros EMP uns aos outros; nem existe um que conecte todas as ocorrências de registros DEPT. Por enquanto, vamos restringir nossa atenção somente aos departamentos. Caso algum programa precise de um caminho de acesso ligando todas as ocorrências DEPT, este caminho poderá naturalmente ser acrescentado. No entanto, o uso do método de montagem de conjunto que descrevemos até agora para esta finalidade seria algo desajeitado. Teríamos que introduzir outro tipo de registro, digamos DEPTS_OWNER, e torná-lo dono de um tipo de conjunto, digamos DEPTSET, com DEPT como membro. Haveria exatamente uma ocorrência de DEPTS_OWNER — possivelmente sem item de dados nela — e uma ocorrência de DEPTSET, sendo cada ocorrência DEPT um membro desta ocorrência DEPTSET singela.

Os *conjuntos singulares* oferecem uma solução mais conveniente ao problema. Um conjunto singular pode ser visto como um conjunto que possui exatamente uma ocorrência sem ter registro dono (no esquema, o dono do conjunto aparece declarado como SYSTEM, ao invés de ter o nome de algum tipo de registro). No exemplo, portanto, poderíamos juntar todas as ocorrências de DEPT em um conjunto único DEPTSET, evitando ter que introduzir o tipo de registro DEPTS_OWNER. De forma semelhante, poderíamos juntar todas as ocorrências de EMP em outro conjunto único denominado EMPSET. Esses dois conjuntos singulares são muito semelhante aos arquivos seqüenciais comuns. EMPSET, por exemplo, fornece acesso seqüencial ao conjunto de todas as ocorrências EMP, de acordo com a ordenação — provavelmente por número de empregado — especificada no esquema para EMPSET. É possível, embora não usual, que um conjunto singular contenha algum *subconjunto* das ocorrências do seu tipo de registro membro (diferindo do arquivo seqüencial normal). Discutiremos esta possibilidade na Seção 24.6.

24.5 UM EXEMPLO DE ESQUEMA

Nesta seção apresentaremos um possível esquema para o banco de dados de fornecedores e peças do Capítulo 4. Voltaremos àquele exemplo para tornar mais fácil uma comparação direta entre as abordagens relacional e de rede. A Fig. 24.13 mostra os dados de exemplo (como na Fig. 4.7), redesenhados para enfatizar a estrutura em rede. As linhas representam as conexões entre fornecedores e peças (tuplas SP na terminologia relacional), e os números que as acompanham representam as quantidades apropriadas.

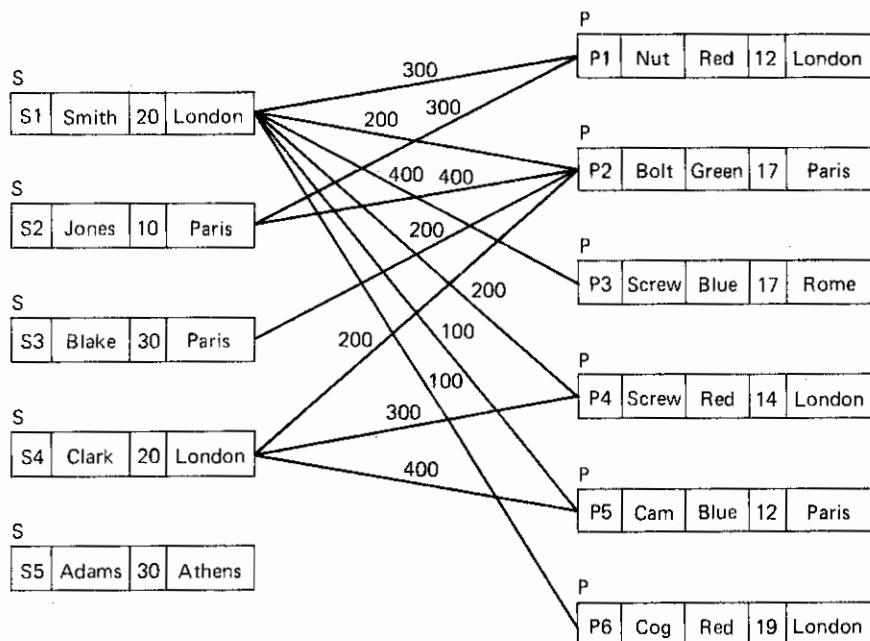


Fig. 24.13 Dados de exemplo (fornecedores e peças).

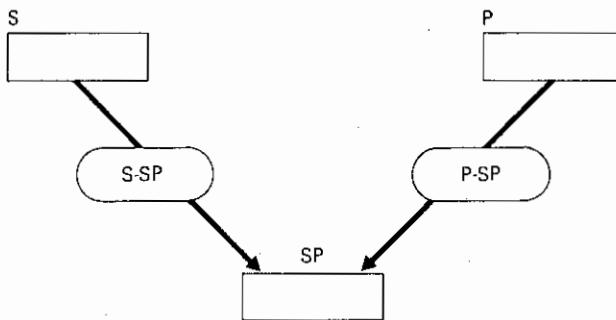


Fig. 24.14 Estrutura dos conjuntos S-SP e P-SP.

Escolhemos uma estrutura em rede para representar esta informação, introduzindo um registro de conexão SP (com itens de dados SNO, PNO, e QTY), e dois conjuntos: S-SP (dono S, membro SP) e P-SP (dono P, membro SP).⁶ O diagrama da estrutura de dados para esta rede está mostrado na Fig. 24.14. Parte do banco de dados correspondente aos dados de exemplo está mostrada na Fig. 24.15.

Ao invés de tentar desenhar todo o banco de dados no estilo da Fig. 24.15, vamos tomar a Fig. 24.13 como uma representação simplificada a partir deste ponto. As linhas naquele diagrama têm portanto que ser consideradas como ocorrências do registro SP. Observe que cada ocorrência de registro SP inclui explicitamente os valores apropriados SNO e PNO; isto é, a estrutura inclui um grau de redundância (o valor SNO de uma dada ocorrência SP poderia sempre ser encontrado no dono da ocorrência do conjunto S-SP na qual aparece a ocorrência SP, o mesmo acontecendo com o valor PNO). Esta redundância foi deliberadamente introduzida, para nos permitir ordenar as ocorrências SP pelos PNO dentro de cada ocorrência S-SP, e pelos SNO dentro de cada ocorrência P-SP. (Teria sido possível evitar a redundância, isto é, excluir os SNO e PNO do tipo de registro SP, mas aí não seria então possível ter o DBMS fazendo a manutenção automática dessas ordenações.)

O esquema está mostrado na Fig. 24.16.

Explicação

A linha 1 designa o nome do esquema.

A linha 3 define a existência do tipo de registro S.

As linhas 4 e 5 especificam que duas ocorrências do tipo de registro S não podem conter o mesmo valor para o item de dado SNO (em um determinado momento). Poderíamos, adicionalmente, especificar que DUPLICATES ARE NOT ALLOWED FOR SNAME IN S, se quiséssemos. Os itens de dados para os quais não se permitem duplicatas são frequentemente usados como base para a localização de registros específicos. Por exemplo, para se localizar a ocorrência de registro S do fornecedor S1, o programador pode forne-

⁶ Usamos SNO e PNO ao invés de S# e P#, porque # não é um caractere válido na DDL esquema.

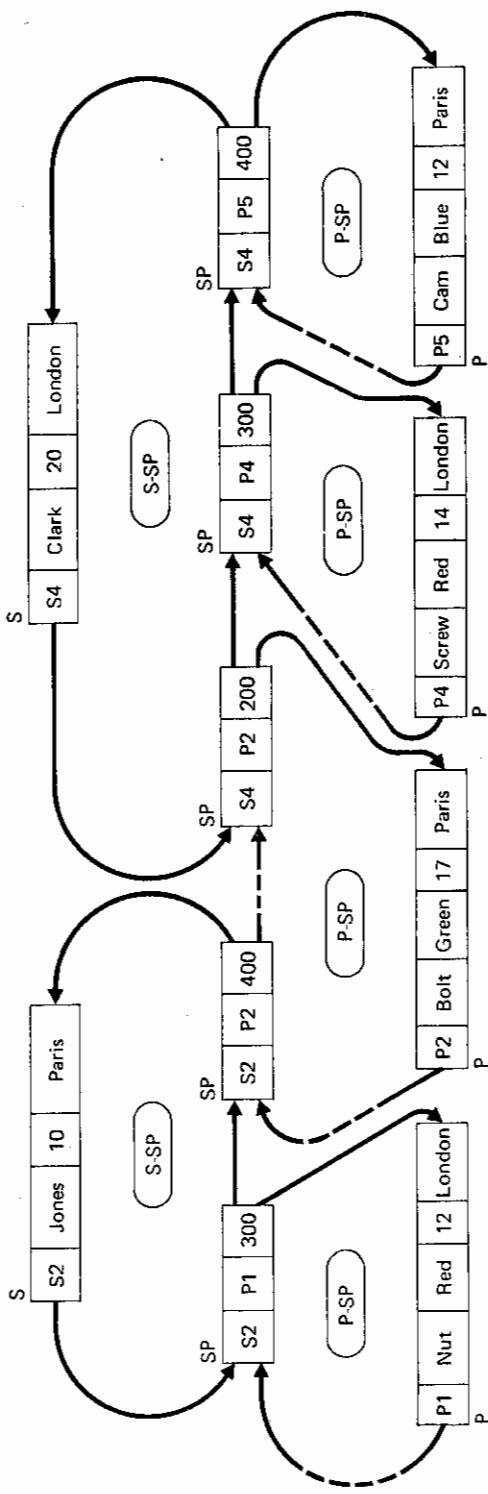


Fig. 24.15 (Parte do) banco de dados de fornecedores e peças.

```

1 SCHEMA NAME IS SUPPLIERS-AND-PARTS.
2
3 RECORD NAME IS S;
4   DUPLICATES ARE NOT ALLOWED
5     FOR SNO IN S.
6     SNO      ; TYPE IS CHARACTER 5.
7     SNAME    ; TYPE IS CHARACTER 20.
8     STATUS   ; TYPE IS FIXED DECIMAL 3.
9     CITY     ; TYPE IS CHARACTER 15.
10
11 RECORD NAME IS P;
12   DUPLICATES ARE NOT ALLOWED
13     FOR PNO IN P.
14     PNO      ; TYPE IS CHARACTER 6.
15     PNAME    ; TYPE IS CHARACTER 20.
16     COLOR    ; TYPE IS CHARACTER 6.
17     WEIGHT   ; TYPE IS FIXED DECIMAL 4; DEFAULT IS -1.
18     CITY     ; TYPE IS CHARACTER 15.
19
20 RECORD NAME IS SP;
21   DUPLICATES ARE NOT ALLOWED
22     FOR SNO IN SP, PNO IN SP.
23     SNO      ; TYPE IS CHARACTER 5.
24     PNO      ; TYPE IS CHARACTER 6.
25     QTY      ; TYPE IS FIXED DECIMAL 5.
26
27 SET NAME IS S-SP;
28   OWNER IS S;
29   ORDER IS SORTED BY DEFINED KEYS
30     DUPLICATES ARE NOT ALLOWED.
31   MEMBER IS SP;
32     INSERTION IS AUTOMATIC
33     RETENTION IS FIXED;
34     KEY IS ASCENDING PNO IN SP;
35     SET SELECTION IS BY VALUE OF SNO IN S.
36
37 SET NAME IS P-SP;
38   OWNER IS P;
39   ORDER IS SORTED BY DEFINED KEYS
40     DUPLICATES ARE NOT ALLOWED.
41   MEMBER IS SP;
42     INSERTION IS AUTOMATIC
43     RETENTION IS FIXED;
44     KEY IS ASCENDING SNO IN SP;
45     SET SELECTION IS BY VALUE OF PNO IN P.

```

Fig. 24.16 O esquema FORNECEDORES-E-PEÇAS.

cer o valor aplicável (isto é, 'S1') ao item de dado SNO IN S, movendo aquele valor para a localização do item de dado com aquele nome na User Work Area, emitindo depois (uma forma apropriada da) instrução DML FIND:

```
MOVE 'S1' TO SNO IN S  
FIND ANY S USING SNO IN S
```

(No Capítulo 26 discutiremos outras maneiras de localizar ocorrências de registros. Observe que na instrução DML "FIND ANY *r* USING *x*", não é exigido que o item de dado *x* seja declarado com DUPLICATES NOT ALLOWED, embora freqüentemente o seja. Veja a Seção 26.11.1 para uma discussão mais detalhada sobre este ponto.)

As linhas 6–9 definem os tipos de itens que constituem S.

As linhas 11–18 definem semelhantemente o tipo de registro P. Observe na linha 17 a frase "DEFAULT IS -1" para o item de dado WEIGHT. O significado desta especificação é que, se WEIGHT for omitido de um registro subesquema correspondente ao registro esquema P, e se o programa criar uma ocorrência do registro P usando aquele subesquema, então o valor de WEIGHT naquela ocorrência será ajustado para -1. De forma geral, se R é um tipo de registro subesquema e devam ser criadas ocorrências de R por algum programa usando aquele subesquema, então R tem que incluir todos os itens de dados do tipo de registro do esquema básico que não tenham uma especificação DEFAULT (valor assumido).

As linhas 20–25 definem de forma semelhante o tipo de registro SP; observe neste caso que foi especificado DUPLICATES NOT ALLOWED para a combinação de dois itens de dados distintos.

A linha 27 define a existência do tipo de conjunto S-SP.

A linha 28 define a ordenação de ocorrências SP dentro de cada ocorrência do conjunto S-SP como sendo SORTED BY DEFINED KEYS. A "chave de controle da ordenação" é definida pela cláusula KEY – linha 34 – como sendo ASCENDING PNO IN SP. Em outras palavras, para cada ocorrência de registro S, as ocorrências de registros SP na correspondente ocorrência do conjunto S-SP encontram-se em ordem ascendente por número de peça. O DBMS é responsável pela manutenção desta seqüência durante a vida do banco de dados. (São permitidas diversas outras formas de ordenação; para detalhes, veja [23.4].) A linha 30 (DUPLICATES ARE NOT ALLOWED) especifica que duas ocorrências SP dentro de uma ocorrência S-SP não podem conter o mesmo valor PNO.

A linha 31 especifica o tipo do registro membro de S-SP, que é SP.

As linhas 32–33 especificam a classe de membro de SP dentro de S-SP. A Seção 24.6 apresenta uma explicação sobre classe de membro.

A linha 34 já foi explicada. De forma geral, a cláusula KEY pode especificar ASCENDING ou DESCENDING, e a "chave" envolvida pode ser a combinação de qualquer quantidade de itens de dados do registro; pode ser especificada uma mistura de chaves ascendentes e descendentes por meio de uma seqüência de entradas ASCENDING/DESCENDING na cláusula KEY. A ordem de especificação dos itens de dados da esquerda para a direita (dentro de uma entrada ASCENDING/DESCENDING ou em diversas) tem o significado usual de maior-para-menor na ordenação.

A linha 35 lida com SET SELECTION, cuja explicação será dada na Seção 24.7.

As linhas 37–45 definem de maneira semelhante o tipo de conjunto P-SP.

24.6 CLASSE DE MEMBRO

Cada subentrada MEMBER no esquema tem que incluir uma especificação sobre a classe de membro do tipo de registro relativo ao tipo de conjunto envolvido. A classe de membro é especificada por meio da entrada INSERTION/RETENTION, podendo portanto ser

vista como uma combinação de uma classe de inserção e uma classe de retenção. A classe de inserção é AUTOMATIC ou MANUAL. A classe de retenção é FIXED, MANDATORY, ou OPTIONAL. Um dado tipo de registro pode ter qualquer combinação de classe de inserção e classe de retenção em relação a um determinado tipo de conjunto; além disso, pode ter diferentes combinações — isto é, diferentes classes de membro — em tipos de conjuntos diferentes. De forma geral, a classe de membro de um registro em um conjunto afeta os programas que lidam com a manutenção daquele conjunto — isto é, programas que criam, modificam ou removem instâncias do relacionamento hierárquico que o conjunto representa. Especificamente, fica afetada a interpretação das instruções DML CONNECT, DISCONNECT, RECONNECT, STORE, e ERASE (e possivelmente MODIFY).

Para fixar a idéia, vamos considerar um conjunto típico OM (dono O, membro M). A Fig. 24.17 mostra uma ocorrência deste conjunto.

- Retention class (FIXED or MANDATORY or OPTIONAL)

Se a classe de M em OM for FIXED, então uma vez que tenha sido entrada uma ocorrência M (digamos m) em uma ocorrência de OM, ela não poderá existir no banco de dados a não ser como membro daquela ocorrência de OM. Especificamente, não poderá ser retirada da ocorrência OM por meio de uma operação DISCONNECT, nem transferida de uma ocorrência OM para outra por meio de uma operação RECONNECT. A única forma de destruir a associação entre m e OM é pela remoção total de m do banco de dados, por meio de uma operação ERASE. Observe aqui a implicação de que se uma ocorrência de O for removida (apagada), todas as ocorrências correspondentes de M terão também que ser removidas.

Se a classe de M em OM for MANDATORY, então uma vez que tenha sido entrada uma ocorrência de M (digamos m) em uma ocorrência de OM, ela não poderá ter existência no banco de dados a não ser como membro de *alguma* ocorrência de OM. Especificamente, não poderá ser retirada da ocorrência OM por meio de operação DISCONNECT, mas poderá ser transferida de uma ocorrência OM para outra por meio de uma operação RECONNECT.

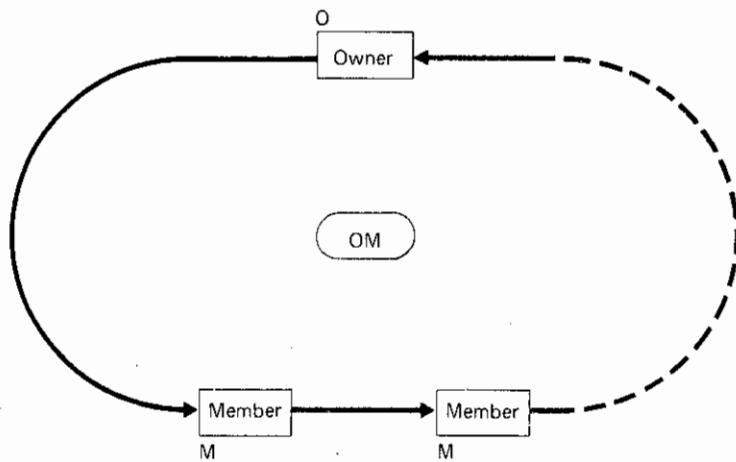


Fig. 24.17 Ocorrência de conjunto típico.

Finalmente, se a classe de M em OM for OPTIONAL, uma ocorrência de M *poderá* ser removida de uma ocorrência de OM (por exemplo, por meio de uma operação DISCONNECT) sem ser totalmente apagada do banco de dados.

Como um exemplo, consideremos o conjunto singular EMPSET, o conjunto de todos os empregados dado na Seção 24.4. Como, por definição, nunca deveremos ter um empregado no banco de dados que *não* seja membro deste conjunto, parece-nos razoável fazer este membro ter classe FIXED (ou MANDATORY – as duas são equivalentes no caso de um conjunto singular). Por outro lado, consideremos o conjunto CONCW (dono CONCERT, membro WORK) do Exemplo 24.3.1. Uma ocorrência deste conjunto representa o programa planejado para um concerto específico. Se o programa para aquele concerto for mudado, terá que ser retirada de CONCW uma ocorrência WORK que não será colocada em nenhuma outra (se aquele trabalho saiu totalmente do programa). Entretanto, ainda pode ser desejável reter a ocorrência WORK no banco de dados, com base em que ela poderá ser entrada em outra ocorrência CONCW mais tarde. Assim, parece ser razoável neste caso tornar a classe deste membro OPTIONAL.

- Insertion class (AUTOMATIC or MANUAL)

Se a classe de M em OM for AUTOMATIC, então quando uma ocorrência de M (*digamos m*) for inicialmente criada e armazenada no banco de dados (por meio de uma operação STORE), o DBMS irá automaticamente conectá-la à ocorrência OM appropriada. (Geralmente cabe ao programa que armazena *m* especificar a ocorrência OM envolvida; veja a Seção 24.7.) Por outro lado, se a classe de M em OM for MANUAL, o armazenamento de uma ocorrência *m* não causará esta conexão automática; para conectar *m* a uma ocorrência OM, o programa tem que emitir uma operação CONNECT explícita.

Como um exemplo, consideremos novamente o conjunto singular EMPSET da Seção 24.4. Uma vez que, como já mencionado anteriormente, nunca teremos um empregado no banco de dados que *não* seja membro deste conjunto, parece ser razoável tornar a classe AUTOMATIC. Por outro lado, consideremos novamente o conjunto CONCW (dono CONCERT, membro WORK) do exemplo 24.3.1. Ali, pode ser uma exigência a capacidade de se armazenar uma ocorrência WORK em particular no banco de dados sem conectá-la imediatamente a alguma ocorrência CONCERT, com a possibilidade de, mais tarde, conectar-a a uma ocorrência do conjunto CONCW, quando o trabalho for selecionado para o programa daquele concerto. Assim, parece ser razoável, neste caso, usar a classe MANUAL.

A Fig. 24.18 representa um resumo parcial do que vimos. A aparente anomalia nessa tabela – CONNECT válido quando a classe é OPTIONAL/AUTOMATIC⁷ – explica-se pelo fato de que, embora venha a ocorrer uma conexão automática quando da execução do STORE, a ocorrência do registro pode mais tarde sofrer um DISCONNECT e um subsequente CONNECT novamente (possivelmente em uma ocorrência diferente do conjunto). O ponto é que, enquanto a classe de retenção (FIXED/MANDATORY/OPTIONAL) é uma propriedade dependente do tempo, a classe de inserção (AUTOMATIC/MANUAL) só tem significado quando a ocorrência do registro envolvida está sendo criada (isto é, no momento do STORE); a partir daí é irrelevante.

7

Na verdade, esta não é a única anomalia. A tabela estaria mais precisa se mostrasse RECONNECT como válido mesmo quando a classe é FIXED. RECONNECT é uma operação legal neste caso, mas somente se a ocorrência de conjunto da qual a ocorrência de registro está sendo desconectado e a ocorrência de conjunto para a qual está sendo transferido forem um e o mesmo (verificado em tempo de execução).

	AUTOMATIC	MANUAL
FIXED	CONNECT X DISCONNECT X RECONNECT X	CONNECT ✓ DISCONNECT X RECONNECT X
MANDATORY	CONNECT X DISCONNECT X RECONNECT ✓	CONNECT ✓ DISCONNECT X RECONNECT ✓
OPTIONAL	CONNECT ✓ DISCONNECT ✓ RECONNECT ✓	CONNECT ✓ DISCONNECT ✓ RECONNECT ✓

Fig. 24.18 Efeito da classe de membro sobre CONNECT, DISCONNECT e RECONNECT.

24.7 SELEÇÃO DE CONJUNTO

Há algumas situações em que o DBMS precisa ser capaz de selecionar automaticamente uma ocorrência específica de um conjunto. Uma dessas situações já foi mencionada na Seção 24.6: uma nova ocorrência deve ser armazenada no banco de dados, e o tipo de registro envolvido é um membro AUTOMATIC de um ou mais tipos de conjuntos. Neste exemplo, o DBMS tem que selecionar a ocorrência apropriada de cada tipo de conjunto aplicável, para poder conectar o novo registro àquela ocorrência de conjunto. Encontraremos outras situações exigindo este processo de seleção automática no Capítulo 26.

Para permitir que o DBMS efetue esta seleção automática de uma ocorrência de conjunto quando necessário, o DBA tem que definir uma cláusula SET SELECTION dentro da entrada do conjunto no esquema (na subentrada MEMBER). Esta seção consiste de uma explicação algo simplificada desta cláusula. Para fixar idéias, vamos restringir nossa atenção ao conjunto S-SP (veja a Seção 24.5), e consideremos a situação em que seja exigido o armazenamento da nova ocorrência SP 'S5/P6/700'. (SP está declarado no esquema — Fig. 24.16 — como sendo um membro AUTOMATIC de S-SP; naturalmente, é também um membro AUTOMATIC de P-SP, mas ignoraremos esse conjunto por simplicidade.)

O caso mais simples da cláusula SET SELECTION para o conjunto S-SP (membro SP) é

```
SET SELECTION IS BY APPLICATION
```

Isto significa simplesmente que o programa de aplicação é responsável pelo procedimento de selecionar a ocorrência S-SP correta antes de armazenar a nova ocorrência SP. Normalmente fará isso procurando (via FIND) a ocorrência correta do dono (tipo de registro S), mas esta etapa pode não ser necessária se a ocorrência corrente S-SP já for a desejada. Em outras palavras, o DBMS simplesmente assumirá que a ocorrência corrente de S-SP é a correta. Para armazenar a ocorrência SP 'S5/P6/700', portanto, uma possível sequência de código é

```
monte a ocorrência 'S5/P6/700' na UWA  
MOVE 'S5' TO SNO IN S  
FIND ANY S USING SNO IN S  
STORE SP
```

Naturalmente, esta é somente uma das muitas possíveis formas de se estabelecer como corrente a ocorrência de S-SP requerida.⁸

O segundo caso é o ilustrado no esquema da Fig. 24.16:

```
SET SELECTION IS BY VALUE OF SNO IN S
```

Isto significa que, quando o DBMS vai selecionar uma ocorrência S-SP, ele deve fazê-lo localizando a ocorrência correspondente do seu dono (S), usando o item de dado SNO IN S (que terá que ter sido especificado com DUPLICATES NOT ALLOWED na definição daquele dono). Isto, por sua vez, significa que o programador tem que inicializar corretamente o item de dado na UWA antes de armazenar a nova ocorrência SP, por exemplo como se segue:

```
monte a ocorrência 'S5/P6/700' na UWA  
MOVE 'S5' TO SNO IN S  
STORE SP
```

Mencionaremos a terceira forma de SET SELECTION apenas para completar o trabalho. Está claro no nosso exemplo que o tipo de conjunto S-SP tem que satisfazer à *restrição estrutural* de que, para qualquer dada ocorrência do conjunto, o valor de SNO IN SP em todos os membros daquela ocorrência tem que ser idêntico ao valor de SNO IN S no dono daquela ocorrência. (De fato, é possível declarar-se esta restrição, passando o DBMS a exigir-la, por meio de uma cláusula especial CHECK, dentro da entrada do conjunto. Esta cláusula não aparece na Fig. 24.16.) Como o conjunto S-SP satisfaz a esta restrição (independentemente de sua declaração via cláusula CHECK), podemos especificar SET SELECTION como

```
SET SELECTION IS BY STRUCTURAL SNO IN SP = SNO IN S
```

significando (neste caso) que o DBMS irá selecionar uma ocorrência do conjunto S-SP através a seleção de uma ocorrência de registro S que tenha um valor de SNO igual ao valor no registro SP sendo armazenado — isto é, igual ao valor de SNO IN SP dentro da UWA. (O item de dado SNO IN S tem que ter DUPLICATES NOT ALLOWED para que esta forma de SET SELECTION seja legal.) A codificação para criar a ocorrência SP 'S5/P6/700' agora se reduz a

```
monte a ocorrência 'S5/P6/700' na UWA  
STORE SP
```

8

O mais recente documento de trabalho ANS sobre a DDL na verdade não permite uma cláusula SET SELECTION especificando BY APPLICATION. Ao invés, BY APPLICATION é simplesmente assumido caso nada seja especificado no esquema, e nada seja especificado também no subesquema (veja o Capítulo 25).

Não discutiremos o caso STRUCTURAL, nem restrições estruturais neste livro.⁹ Para detalhes veja [23.3], [23.4].

Finalmente, observemos que a cláusula SET SELECTION não é necessária e nem permitida caso o conjunto envolvido seja singular. Com efeito, a ocorrência única do conjunto é sempre considerada como a ocorrência corrente.

O processo de SET SELECTION não tem efeito sobre os indicadores de posição corrente em uma corrida do programa (indicadores de posição corrente estão discutidos no Capítulo 26).

EXERCÍCIOS

24.1 Defina um esquema para uma versão DBTG do banco de dados de publicações do Exercício 16.2. (Mantenha a estrutura hierárquica, isto é, não procure convertê-la em uma rede.)

24.2 Defina um esquema (em esboço) para o banco de dados de "estrutura gerencial" da Seção 24.2.4.

24.3 Defina um esquema para uma versão DBTG do banco de dados de fornecedores-peças-projetos. *Nota:* Vários exercícios no Capítulo 26 estarão baseados neste esquema.

24.4 Defina um esquema (em esboço) para o banco de dados de peças e componentes da Seção 24.3.3.

24.5 Considere o banco de dados de áreas e pássaros da Seção 20.7. Defina um esquema para uma versão DBTG (a) como uma "rede de duas entidades", envolvendo AREAs e BIRDs; (b) como uma "rede de três entidades" envolvendo AREAs, BIRDs, e DATEs. (Observe que a situação é fundamentalmente uma de "três entidades", pois uma dada combinação AREA-BIRD pode ocorrer em várias DATEs. Pode ser assumido que não aparece no banco de dados mais do que uma observação por dia de cada tipo de pássaro dentro de cada área.)

24.6 Projete um banco de dados DBTG para representar uma rede de transporte. Pode tomar como base para o projeto de sua rede alguma que lhe seja familiar – por exemplo, o metrô do Rio.

REFERÊNCIAS E BIBLIOGRAFIA

Veja também [23.4].

24.1 C. W. Bachman. "Data Structure Diagrams". *Data Base* (journal of ACM SIGBDP) 1, nº 2 (Verão de 1969).

24.2 C. W. Bachman. "Implementation Techniques for Data Structure Sets." In reference [1.12].

Descreve uma série de técnicas possíveis para mapear o conjunto DBTG na memória. São incluídas características de desempenho.

24.3 B. C. M. Douqué and G. M. Nijssen (eds.). "Data Base Description." *Proc. IFIP TC-2 Special working Conference on Data Base Description* (janeiro de 1975). North-Holland (1975).

9

Um conjunto com uma restrição estrutural declarada é uma aproximação do que, no Capítulo 28, nós chamaremos de "conjunto não essencial". Como o exemplo ilustra, o objetivo geral dos conjuntos não essenciais é o de tornar mais fácil a manutenção do banco de dados em um estando adequado de integridade, de forma que, por exemplo, um registro membro não venha a ser conectado ao dono errado. Mas o contexto total da interação entre o conceito de não-essencialidade, por um lado, e SET SELECTION, classe de membro, e semântica dos operadores de atualização (STORE, ERASE, MODIFY, CONNECT, DISCONNECT, RECONNECT), por outro lado, é muito complexo. (Como um exemplo simples, considere o efeito de um MODIFY sobre uma instância do item de dado SNO IN SP, se o conjunto S-SP for não essencial.) Além disso, como questionaremos no Capítulo 28 e como o próprio nome sugere, há boas razões para não se incluir conjuntos não essenciais no esquema.

O objetivo principal desta conferência foi o de fazer uma avaliação crítica da DDL esquema CODASYL. No final da conferência, os participantes formularam um conjunto de recomendações para melhoria da DDL. As recomendações estão resumidas abaixo.

- Permitir que um dado tipo de conjunto tenha o mesmo tipo de registro, tanto como dono quanto como membro.
- Eliminar grupos repetidos.
- Permitir a especificação de qualquer quantidade de identificadores para um determinado tipo de registro. (Este dispositivo já foi incorporado na DDL, via cláusula DUPLICATES NOT ALLOWED.)
- Permitir a especificação de SEARCH KEY para um dado tipo de registro, para permitir a otimização do acesso a ocorrências de registros na base de valores de itens de dados especificados.
- Permitir acesso (da DML) a ocorrências de registros na base de valores especificados de itens de dados. (A instrução FIND agora fornece essa função.)
- Permitir que SET SELECTION seja baseado em outros identificadores além das CALC-keys. (Este dispositivo já foi efetivamente incorporado à DDL. As CALC-keys desapareceram.)
- Permitir somente um tipo de registro membro por tipo de conjunto.
- Consolidar SEARCH KEY e SORTED INDEXED. (Já feito.)
- Melhorar a capacidade de seleção de SET SELECTION – em particular, suporte para quantificação existencial.
- Permitir que sejam especificadas restrições de cardinalidade na declaração do conjunto (quantidade de ocorrências de membros por ocorrência de dono).

24.4 H. Schenk. "Implementational Aspects of the CODASYL DBTG Proposal." In "Data Base Management" (eds. Klimbie and Koffeman), *Proc. IFIP TC-2 Working Conference on Data Base Management Systems* (abril de 1974). North-Holland (1974).

Inclui uma descrição de como os registros e conjuntos são implementados em um sistema DBTG chamado PHOLAS. Discute também o manuseio do UWA, buffers do sistema, e os formatos objeto do esquema e subesquema.

24.5 R. Gerritsen. "A Preliminary System for the Design of DBTG Data Structures." *CACM* 18, nº 10 (outubro de 1975).

Descreve um método automático (implementado sob a forma de um programa chamado o Designer) de geração de um projeto de banco de dados, dado um conjunto de consultas antecipadas. Esta técnica é conhecida como uma abordagem funcional ao projeto de banco de dados, em contraste com a abordagem "existencial" de se modelar a empresa de uma forma independente dos usos a serem feitos da informação. Como um exemplo simples, a consulta "liste todos os fornecedores que fornecem a peça P1" faz com que o Designer gere "ABOVE (SUPPLIER PART)" – significando que os registros de fornecedores ficariam provavelmente em posição hierarquicamente superior à dos registros de peças, na estrutura resultante.

24.6 BCS/CODASYL DDLC Data Base Administration Working Group. "Draft Specification of a Data Storage Description Language." Apêndice de [23.4].

Propõe uma linguagem (DSDL) para a definição do esquema de armazenamento. Os aspectos principais dessa linguagem estão resumidos adiante.

- O espaço total de armazenamento fica particionado em *áreas separadas*, cada uma consistindo de um número integral de *páginas*. O tamanho da página é fixo dentro de uma área de armazenamento, mas pode variar de uma área para outra. O tamanho da área pode ser fixo ou variável.
- Um tipo de registro definido ao nível de esquema fica representado por um ou mais *tipos de registros armazenados* ao nível do esquema de armazenamento. Por exemplo, o tipo de registro S de fornecedor pode ser mapeado a dois tipos de registros armazenados SX e SY,

com SX contendo itens de dados SNO, SNAME, e CITY, e SY contendo SNO e STATUS. As ocorrências SX e SY correspondentes a uma dada ocorrência S são encadeadas (a duplicação de SNO é opcional). É também possível particionar ocorrências do tipo de registro esquema; por exemplo, algumas ocorrências S (digamos aquelas com CITY = 'LONDON') podem ser mapeadas a um tipo de registro armazenado, e outras para outro.

Todas as ocorrências de um determinado tipo de registro de armazenamento são armazenadas na mesma área de armazenamento. A colocação dentro da área pode ser seqüencial, "CALC" (randomizado), ou "grupado", de acordo com um tipo de conjunto especificado (por exemplo, os registros de embarque poderiam ser armazenados próximo aos correspondentes registros de fornecedores).

- Os tipos de esquemas de conjunto são representados por cadeias de indicadores de localização embutidas ou índices. No caso de índice, o índice consiste de uma coleção de "subíndices" (não é um termo DSDL), um para cada ocorrência do conjunto sendo representada, consistindo cada subíndice de um índice da ocorrência do registro membro dentro daquela ocorrência de conjunto. Cada ocorrência de registro dono indica o subíndice correspondente.
- Os índices podem também ser usados para fornecer caminhos adicionais de acesso não expostos no esquema. A colocação do índice dentro das áreas de armazenamento fica sob controle do projetista do esquema de armazenamento.

25

O Nível Externo do DBTG

25.1 INTRODUÇÃO

Como explicamos no Capítulo 23, uma visão externa em DBTG é definida por meio de um *subesquema*. Neste capítulo, examinaremos o subesquema COBOL. *Grosso modo*, o subesquema é simplesmente um subconjunto do esquema correspondente; detalhes mais específicos sobre as diferenças existentes entre um dado subesquema e o esquema correspondente encontram-se na Seção 25.2.

Pode ser definida qualquer quantidade de subesquemas para um determinado esquema; qualquer quantidade de programas pode compartilhar um dado subesquema; subesquemas diferentes podem se sobrepor. Lembre-se de que é a “invocação” do subesquema por um programa que dá àquele programa a definição da sua User Work Area (UWA). Um exemplo dessa invocação é

DB SUPPLIERS WITHIN SUPPLIERS-AND-PARTS.

Esta instrução (que aparece na Data Division do programa COBOL) invoca o subesquema chamado SUPPLIERS, que é derivado do esquema chamado SUPPLIERS-AND-PARTS. Mostraremos uma possível definição de SUPPLIERS na Seção 25.3.

25.2 DIFERENÇAS ENTRE O SUBESQUEMA E O ESQUEMA

Tornemos inicialmente mais precisa a afirmativa de que um subesquema é um subconjunto do esquema. De forma geral, qualquer das seguintes entradas de esquema pode ser omitida em um dado subesquema.

- A declaração de um ou mais conjuntos
- A declaração de um ou mais registros
- A declaração de um ou mais itens de dados

O subesquema tem que ser, naturalmente, autoconsistente. Por exemplo, uma declaração de registro não poderá ser omitida se for incluída uma declaração de conjunto de um tipo de conjunto em que aquele tipo de registro seja o dono.

Ocorre então que, como no IMS, os usuários estão protegidos de certos tipos de crescimento no esquema. Por exemplo, pode ser adicionado um novo tipo de item de dado a um tipo de registro existente. De forma semelhante, podem ser adicionados novos tipos de registros. Sob certas circunstâncias, podem ser adicionados novos tipos de conjuntos. (Veja o Exercício 25.1).

Ocorre também que o subesquema fornece automaticamente um nível de segurança de dados, na medida em que um programa não poderá ter acesso a qualquer dado não definido no esquema correspondente (exceto no caso de ERASE; veja o Capítulo 26).

Outras diferenças importantes que podem existir entre o esquema e o subesquema são:

- Podem ser definidos nomes privativos (“aliases”) para conjuntos, registros, e itens de dados.
- Os itens de dados podem ter tipos de dados diferentes.
- Pode ser modificada a ordem relativa dos itens de dados dentro dos registros que os contêm.
- Os conjuntos podem receber cláusulas SET SELECTION diferentes. Destas, somente a última parece necessitar de explicação. A idéia é que a cláusula SET SELECTION no esquema possa ser modificada pela do subesquema. Por exemplo, o esquema poderia especificar BY APPLICATION, significando que a seleção daquela ocorrência de conjunto deva ser feita totalmente por procedimento, enquanto que um subesquema individual poderia especificar BY VALUE OF, de modo que, para programas que invoquem aquele subesquema específico, a operação possa ser menos por procedimentos.

25.3 UM EXEMPLO DE SUBESQUEMA

A Fig. 25.1 consiste de um exemplo simples de um subesquema COBOL baseado no esquema da Fig. 24.16. Basicamente, este subesquema inclui informações sobre fornecedores e embarques do esquema básico, mas exclui informações sobre peças.

De forma geral, um subesquema COBOL consiste de uma Title Division (que dá nome ao subesquema e identifica o esquema básico), uma Mapping Division (onde são definidos os aliases), e uma Structure Division. A Structure Division, por sua vez, consiste de três Seções: uma Realm Section, uma Record Section, e uma Set Section. A Realm Section está explicada abaixo. A Record Section lista todos os tipos de registros esquema que interessam (em cada caso, somente com os itens de dados requeridos). A Set Section lista todos os conjuntos esquema que interessam (possivelmente cada um com sua própria reespecificação do SET SELECTION).

A Realm Section especifica todos os *Realms* que interessam. Um realm é um subconjunto lógico da visão total do banco de dados definida pelo subesquema. Consiste de todas as ocorrências de registros de um ou mais tipos especificados (frequentemente, todos os tipos listados na Record Section, como na Fig. 25.1). Um subesquema pode listar

¹ Veja nota 8 de rodapé no Capítulo 24, pág. 416.

TITLE DIVISION.

SS SUPPLIERS WITHIN SUPPLIERS-AND-PARTS.

MAPPING DIVISION.

ALIAS SECTION.

AD SET SUPPLIES IS S-SP.

STRUCTURE DIVISION.

REALM SECTION.

RD SUPPLIES-REALM CONTAINS S, SP RECORDS.

RECORD SECTION.

01 S.

 02 SNO ; PICTURE IS X(5).

 02 CITY; PICTURE IS X(24).

01 SP.

 02 PNO ; PICTURE IS X(6).

 02 SNO ; PICTURE IS X(5).

 02 QTY ; PICTURE IS S9(5).

SET SECTION.

SD SUPPLIES.

Fig. 25.1 Exemplo de subesquema (COBOL).

qualquer quantidade de realms, sendo que realms distintos podem se superpor (ter tipos de registros em comum). Cada realm especifica um “escopo de processamento”; antes que um programa possa ter acesso a qualquer parte do banco de dados definido em seu subesquema, ele tem que tornar READY o realm apropriado (análogo a se abrir um arquivo convencional), para informar ao DBMS sobre sua intenção de ter acesso àquela parte do banco de dados. A instrução READY também indica se o acesso irá se limitar a operação do tipo RETRIEVAL ou incluirá operações do tipo UPDATE. Maiores detalhes encontram-se no Capítulo 26.

A partir deste ponto, e em particular no Capítulo 26, estaremos supondo que o subesquema é idêntico, em todos os aspectos relevantes, ao esquema correspondente.

EXERCÍCIO

25.1 Suponha que seja adicionado ao esquema um novo tipo de conjunto OM (dono O, membro M). Até que ponto os programas existentes permanecem sem serem afetados por esta adição? Considere cada um dos seguintes casos:

- O e M sendo ambos novos tipos de registros (adições).
- O e M sendo ambos tipos antigos (existentes) de registros.
- O antigo e M novo.
- O novo e M antigo.

(Nota: A classe de membro é importante aqui; SET SELECTION pode ser também. Talvez você prefira adiar as considerações detalhadas sobre este exercício para depois de ter lido o Capítulo 26 e revisto as Seções 24.7 e 24.8.)

REFERÊNCIAS E BIBLIOGRAFIA

Veja [23.3] e [23.4].

- 25.1 M. H. Kay. "An Assessment of the CODASYL DDL for Use with a Relational Subschema." In "Data Base Description" (eds. Douqué and Nijssen), *Proc. IFIP TC-2 Working Conference on Data Base Description* (fevereiro de 1975). North-Holland (1975).

Apresenta algumas idéias sobre o suporte a um subesquema relacional montado sobre um esquema DBTG. De forma geral, tal subesquema é mais do que simplesmente "um simples subconjunto" do esquema. O artigo identifica alguns problemas que surgem, e critica a DDL esquema à luz desses problemas.

- 25.2 C. A. Zaniolo. "Multimodel External Schemas for CODASYL Data Base Management Systems." In "Data Base Architecture" (eds., Bracchi and Nijssen), *Proc. IFIP TC-2 Working Conference on Data Base Architecture* (junho de 1979). North-Holland (1979).

- 25.3 C. A. Zaniolo. "Design of Relational Views over Network Schemas." *Proc. 1979 ACM SIGMOD International Conference on Management of Data*.

- 25.4 L. I. Mercz. "Issues in Building a Relational Interface on a CODASYL DBMS." In "Data Base Architecture" (eds., Bracchi and Nijssen), *Proc. IFIP TC-2 Working Conference on Data Base Architecture* (junho de 1979). North-Holland (1979).

- 25.5 E. K. Clemons. "The External Schema and CODASYL." *Proc. 4th International Conference on Very Large Data Bases* (1978).

- 25.6 E. K. Clemons. "An External Schema Facility for CODASYL 1978." *Proc. 5th International Conference on Very Large Data Bases* (1979).

26

Manipulação de Dados DBTG

26.1 INTRODUÇÃO

Como fizemos com o subesquema DDL, vamos basear nossa explicação sobre a manipulação de dados DBTG na DML definida para COBOL. O planejamento do capítulo é o seguinte. A Seção 26.2 lida com o conceito de posição corrente, cuja compreensão é um pré-requisito para o entendimento de qualquer instrução DML. A Seção 26.3 discute o manuseio de erros e exceções. As Seções 26.4–26.11 lidam com as principais instruções da linguagem; em particular, a instrução FIND é vista na última Seção, 26.11, porque, embora sendo sem dúvida a mais importante das instruções na DML, é também a mais complexa. A Seção 26.12 fornece uma breve descrição das instruções restantes.

Todos os exemplos serão baseados nos dados da Fig. 24.13 e no esquema da Fig. 24.16 (fornecedores e peças). Como mencionamos antes, estaremos supondo que o subesquema é essencialmente igual ao esquema; a invocação do subesquema é feita como indicamos na Seção 25.1. Por conveniência, os dados de amostragem estão novamente mostrados na Fig. 26.1.

26.2 INDICAÇÃO CORRENTE

Antes de examinarmos as instruções da DML, é essencial discutirmos o conceito fundamental de “corrente”. Este conceito é uma generalização da noção familiar de posição corrente dentro de um arquivo. A idéia básica é que, para cada programa que opera sob seu controle, o DBMS mantém uma tabela de “indicadores de correntes”. Um indicador de corrente é um objetivo cujo valor é (tipicamente) uma *chave do banco de dados*. Chaves do banco de dados são valores gerados pelo sistema que identificam de forma única registros no banco de dados (podem ser vistos como endereços de armazenamento de registros, no sentido do Capítulo 2). Os indicadores de correntes para uma dada unidade-de-

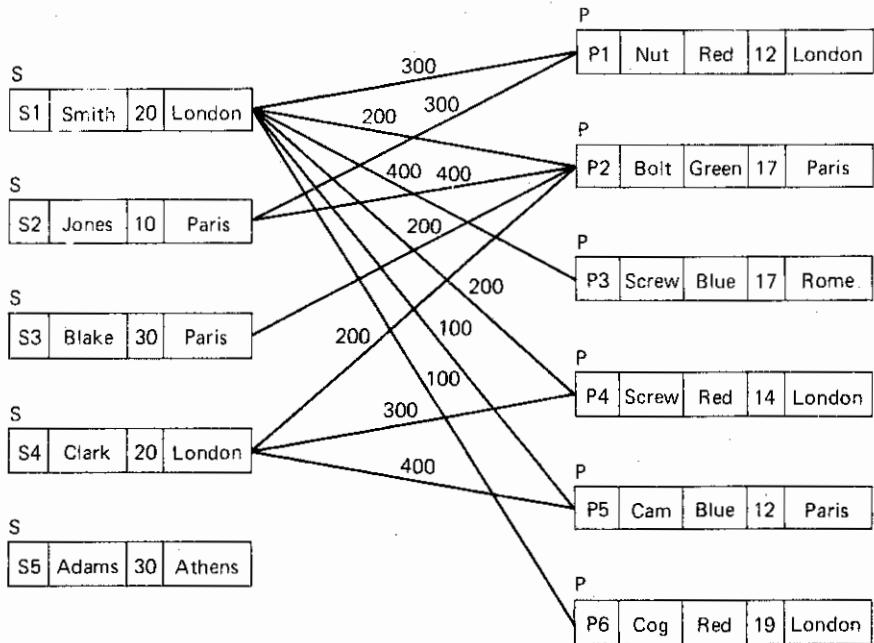


Fig. 26.1 Dados de amostragem (fornecedores e peças).

corrida¹ identificam a ocorrência de registro que recebeu acesso² mais recente nessa unida de de corrida, englobando:

- Cada realm.

Para um realm R, a ocorrência de registro dentro de R que recebeu o acesso mais recente é o “registro corrente do realm R”.

- Cada tipo de registro.

Para um tipo de registro T, a ocorrência de T que recebeu o acesso mais recente é o “registro corrente do tipo T” ou “a ocorrência T corrente”.

- Cada tipo de conjunto.

Para um conjunto S, a ocorrência de registro que recebeu o acesso mais recente e participa de uma ocorrência de S pode ser uma ocorrência de dono ou de membro. Qualquer que seja, é o “registro corrente do conjunto S”. Observe que o “registro corrente do conjunto S” refere-se a uma *ocorrência de registro*, mas que também identifica de forma única uma ocorrência de *conjunto*, ou seja, a ocorrência única de S que o contém; esta ocorrência de conjunto é a “ocorrência corrente de S”.

¹ “Unidade-de-corrida” significa a execução de um programa.

² “Que recebeu acesso mais rececente” não está estritamente preciso aqui; veja a Seção 26.11.7.

- Qualquer tipo de registro

A ocorrência de registro que recebeu acesso mais recente, não importando o seu tipo (e não importando a que realm pertence e de que conjunto participa), é o “registro corrente da unidade de corrida” (usualmente abreviado somente para “o corrente da unidade de corrida”). Este é o “corrente” mais importante de todos, como veremos adiante.

Por exemplo, considere a seguinte sequência de instruções.

```
MOVE 'S4' TO SNO IN S
FIND ANY S USING SNO IN S
FIND FIRST SP WITHIN S-SP
FIND OWNER WITHIN P-SP
```

O efeito destas instruções é o seguinte. O MOVE inicializa a User Work Area — item de dado SNO IN S. O primeiro FIND localiza a ocorrência correspondente de registro S, a do fornecedor S4. O próximo FIND localiza a primeira ocorrência de SP dentro da ocorrência de conjunto S-SP cujo dono é o fornecedor S4, ou seja, a ocorrência SP ‘S4/P2/200’; e o último FIND localiza o dono desta ocorrência SP dentro do conjunto P-SP, ou seja, a ocorrência P da peça P2. Portanto, no final desta sequência, esta ocorrência P é o corrente da unidade de corrida. Complete agora o restante da tabela:

Corrente da unidade de corrida	P 'P2'
Ocorrência S corrente	
Ocorrência P corrente	
Ocorrência SP corrente	
Registro corrente do conjunto S-SP	
Registro corrente do conjunto P-SP	
Ocorrência S-SP corrente	
Ocorrência P-SP corrente	
Registro corrente do realm S-SP-P	

(Estamos supondo que o realm S-SP-P engloba todo o banco de dados. A tabela completa encontra-se na seção de respostas, no final do livro.)

O seguinte resumo das principais instruções DML mostra por que o “corrente” — e particularmente o “corrente da unidade de corrida” — é uma noção tão importante.

- FIND localiza uma ocorrência de registro existente e a torna o corrente da unidade de corrida (atualizando também outros indicadores de corrente, como for apropriado).
- GET recupera o corrente da unidade de corrida.
- ERASE elimina o corrente da unidade de corrida.
- STORE cria uma nova ocorrência de registro e a torna o corrente da unidade de corrida (atualizando também outros indicadores de corrente, como for apropriado).
- MODIFY atualiza o corrente da unidade de corrida.
- CONNECT conecta o corrente da unidade de corrida a uma ocorrência de conjunto.

- DISCONNECT desconecta o corrente da unidade de corrida de uma ocorrência de conjunto.
- RECONNECT desconecta o corrente da unidade de corrida de uma ocorrência de conjunto, e o conecta a outra ocorrência de conjunto do mesmo tipo.

Fica aparente deste resumo a importância da instrução FIND: ela é logicamente exigida antes de cada uma das outras instruções, exceto STORE. Entretanto, como mencionamos na introdução, deixaremos o estudo de FIND para depois das outras, restringindo-nos a formatos da instrução FIND já mostrados, quando esta for necessária.

26.3 MANUSEIO DE EXCEÇÕES

Como etapa final na execução de qualquer instrução DML, o DBMS coloca um valor no "registrar especial do banco de dados" DB-STATUS para indicar o resultado dessa instrução; um valor zero significa que a instrução foi bem-sucedida, e um valor diferente de zero significa que ocorreu uma condição de erro ou de exceção. O programador tem que fornecer um ou mais "procedimentos declarativos" (via a instrução "USE FOR DB-EXCEPTION") para manusear todos os casos diferentes de zero. Esses procedimentos são especificados no início da Procedure Division, usando uma forma simplificada da sintaxe COBOL.

DECLARATIVES.

USE FOR DB-EXCEPTION ON '0502100'.

EOF-PROC.

MOVE 'YES' TO EOF.

USE FOR DB-EXCEPTION ON '0502400'.

NOTFOUND-PROC.

MOVE 'YES' TO NOTFOUND.

USE FOR DB-EXCEPTION ON OTHER.

OTHER-PROC.

END DECLARATIVES.

O primeiro USE especifica um procedimento — consistindo de uma só instrução MOVE 'YES' TO EOF — a ser executada se DB-STATUS receber o valor 0502100 (que ocorre quando a posição corrente de uma operação FIND NEXT for um fim de conjunto ou um fim de realm). O segundo USE especifica um procedimento semelhante para o valor 0502400 de DB-STATUS ("registro não encontrado"). O terceiro especifica um procedimento "não faça nada" (vazio), a ser executado sempre que ocorra qualquer outro valor não-zero de DB-STATUS (isto é, um valor que não seja manuseado pelas instruções USE mais explícitas já mostradas).

De forma geral, a instrução USE pode especificar ou OTHER ou uma lista de valores específicos. Quando uma instrução DML faz com que DB-STATUS seja ajustado para um valor diferente de zero, o procedimento declarativo correspondente é invocado. Tem que haver exatamente um procedimento (possivelmente vazio) correspondente a cada um dos valores. Após a execução do procedimento, o controle é retornado à instrução que se

segue à instrução DML original. (A especificação de um procedimento vazio permite o manuseio da condição correspondente em linha – isto é, a instrução DML pode ser seguida por um teste explícito do valor correspondente do DB-STATUS, e de codificação explícita para manusear a condição correspondente, se necessário).

Em adição a DB-STATUS, alguns outros registradores especiais (DB-SET-NAME, DB-RECORD-NAME, DB-DATA-NAME) são também ajustados após a execução de certas instruções DML. Esses registradores, que têm interpretação óbvia, podem auxiliar no processo de depuração de erros e outras situações.

No que se segue, estaremos normalmente ignorando a necessidade dos testes de exceções.

26.4 GET

- Obtenha todos os detalhes do fornecedor S4.

```
MOVE 'S4' TO SNO IN S
FIND ANY S USING SNO IN S
GET S
```

O efeito desta codificação é o de trazer a ocorrência de registro S do fornecedor S4 para a localização de S dentro da UWA. Observe que a instrução FIND por si *não* recupera dados. Incidentalmente, uma vez que GET (como a maioria das outras instruções) opera sobre o corrente da unidade de corrida, seria perfeitamente válido escrever simplesmente

```
GET
```

Isto traria o corrente da unidade de corrida, qualquer que fosse o seu tipo, para a localização apropriada na UWA. Entretanto, ao especificar um nome de registro (S no exemplo), o programador permite que o DBMS verifique que o corrente da unidade de corrida seja do tipo apropriado (e, de qualquer maneira, essa especificação explícita é mais clara). No exemplo, se por alguma razão o corrente da unidade de corrida não fosse uma ocorrência S, o GET iria falhar, sendo colocado um valor não-zero em DB-STATUS.

A maioria das instruções DML permite que seja omitida a especificação do nome do registro, como no caso do GET. Na prática, entretanto, isto não é recomendado, e nós não o faremos em nossos exemplos.

É também possível obter-se somente os itens de dados especificados do corrente da unidade de corrida, como ilustra o exemplo a seguir:

- Obtenha o nome e a cidade do fornecedor S4

```
FIND S occurrence for S4
GET SNAME IN S, CITY IN S
```

Neste exemplo, somente são levados para a UWA o nome e a cidade do fornecedor S4; os outros itens de dados permanecem inalterados.

26.5 STORE

- Crie a ocorrência SP 'S5/P6/700'.

- MOVE 'S5' TO SNO IN SP
MOVE 'P6' TO PNO IN SP
MOVE 700 TO QTY IN SP
- MOVE 'S5' TO SNO IN S
MOVE 'P6' TO PNO IN P
- STORE SP

A etapa 1 consiste das instruções — três MOVES — que constroem a nova ocorrência SP na UWA. Ela pode agora ser armazenada no banco de dados (etapa 3). Entretanto, SP está declarado no esquema da Fig. 24.16 como sendo um membro AUTOMATIC tanto de S-SP quanto de P-SP; portanto, quando uma ocorrência SP é armazenada, o DBMS tem que automaticamente conectá-la às ocorrências apropriadas desses dois conjuntos — no exemplo, a ocorrência S-SP cujo dono é S5 e a ocorrência P-SP cujo dono é P6. Em geral, as ocorrências S-SP e P-SP às quais tem que ser conectada a nova ocorrência SP são escolhidas pelo DBMS com base nas cláusulas SET SELECTION dadas no esquema para esses dois conjuntos. A Seção 24.7 deu uma explicação completa sobre esta cláusula; neste ponto é suficiente observar que, para que o processo de SET SELECTION possa pegar as ocorrências de conjuntos corretas neste exemplo, o programador tem que primeiramente inicializar — etapa 2 — os itens de dados SNO IN S e PNO IN P na UWA com os valores corretos.

Após o STORE no exemplo, a nova ocorrência SP é o corrente da unidade de corrida, a ocorrência SP corrente, o registro corrente do realm S-SP-P, e o registro corrente de S-SP e P-SP. (Em geral, um registro recém-armazenado torna-se o registro corrente de todos os conjuntos dos quais ele é o dono ou um membro AUTOMATIC, exceto quanto ao que será explicado subsequentemente na Seção 26.11.7.)

26.6 ERASE

- Elimine a ocorrência S do fornecedor S4.

```
FIND S occurrence for S4
ERASE [ ALL ] S
```

Como o exemplo mostra, há dois formatos de ERASE, um com e outro sem a especificação ALL. Seus significados diferem quando o registro alvo (o corrente da unidade de corrida) é o dono de uma ou mais ocorrências de conjuntos não-vazios, como neste caso. Vamos definir o termo “descendente de R”, recursivamente, como significando um registro membro cujo dono em algum conjunto é R ou um descendente de R. A instrução ERASE ALL elimina o corrente da unidade de corrida e todos os seus descendentes; no exemplo, ERASE ALL eliminaria o fornecedor S4 e todos os seus embarques. Por outro lado, a instrução ERASE sem All opera de acordo com o procedimento descrito nas etapas 1—6 abaixo. (O objetivo deste procedimento é o de que ERASE falhe se solicitar a remoção de um registro que tenha membros MANDATORY; cause o *disconnect* de membros OPTIONAL e a remoção de membros FIXED.)

1. Seja R o corrente da unidade de corrida.
2. Se existir um registro X que seja membro MANDATORY de um conjunto T com dono R, então o ERASE falha; são removidas quaisquer indicações deixadas ante-

riormente de “removível” ou “desconectável” durante a operação (etapas 3 e 5 abaixo), e o ERASE termina sem sucesso (DBSTATUS diferente de zero). O banco de dados permanece exatamente no estado em que estava antes do ERASE.

3. Para cada registro X que seja um membro OPTIONAL de um conjunto T com dono R, X recebe a indicação de “desconectável de T”.
4. Para cada registro X que seja um membro FIXED de um conjunto T com dono R, são repetidas as etapas 2–5 com X substituindo R.
5. R recebe a indicação de “removível”.
6. Todos os registros com indicação de “desconectáveis” são desconectados do conjunto indicado. Todos os registros com indicação de “removíveis” são desconectados de todos os conjuntos dos quais são membros e destruídos. A operação ERASE termina com sucesso (DBSTATUS é igual a zero).

Observe que os descendentes podem ser removidos ou desconectados mesmo se não forem visíveis através do subesquema.

Após um ERASE bem-sucedido, o indicador do corrente do “corrente da unidade de corrida” é nulo (não identifica uma posição). O efeito sobre outros indicadores do corrente ocorre como a seguir. Seja R o registro a ser apagado, e suponhamos que R seja o registro corrente de algum realm M, o registro corrente do seu tipo de registro, o registro corrente do conjunto T1 (no qual ele é o dono), e o registro corrente do conjunto T2 (no qual ele é um membro); em outras palavras, todos os indicadores do corrente – para M, para o tipo de registro considerado, para o tipo de conjunto T1 e para o tipo de conjunto T2 – estão designando o registro R. Suponhamos agora que R foi apagado.

- O indicador do corrente do realm M é atualizado para não mais designar um registro, mas sim identificar o “buraco” deixado por R dentro do realm. Uma tentativa para se encontrar o próximo registro depois de R dentro do realm M (usando um FIND formato 3; veja a Seção 26.11.3) irá mover o indicador do corrente do realm M para o registro que se segue a este buraco dentro de M.
- O indicador do corrente para o tipo de registro é atualizado de maneira semelhante; isto é, passa a identificar o buraco. Uma tentativa para se encontrar o próximo registro depois de R usando um FIND formato 1 (Seção 26.11.1) irá mover o indicador do corrente para o registro que se segue ao buraco.
- O indicador do corrente do conjunto T1 é ajustado para nulo.
- O indicador do corrente do tipo de conjunto T2 é atualizado para identificar o buraco. Uma tentativa para se encontrar o próximo registro depois de R dentro do tipo de conjunto T2 (usando um FIND formato 3) irá mover o indicador do corrente para o registro que se segue ao buraco.

26.7 MODIFY

- Adicione 10 ao valor de status do fornecedor S4.

```
FIND S occurrence for S4
GET S
ADD 10 TO STATUS IN S
MODIFY S
```

A instrução MODIFY substitui (partes do) corrente da unidade de corrida por valores retirados da UWA. Portanto, para adicionar 10 ao valor de status de S4, o usuário tem que recuperá-lo, incrementá-lo na UWA, e depois colocá-lo de volta. Se o MODIFY especificar um nome de registro, todo o registro é substituído. Se especificar uma lista de itens de dados, somente são substituídos aqueles itens. No exemplo, portanto, o efeito desejado poderia ser igualmente obtido pela especificação de STATUS IN S (ao invés de simplesmente S) no GET e no MODIFY.

Sob certas circunstâncias – por exemplo, se for mudada uma chave de controle de ordenação –, o MODIFY pode causar a atualização de indicadores do corrente. Veja [23.3] para maiores detalhes.

26.8 CONNECT

Da mesma forma como as instruções STORE, ERASE, e MODIFY são fornecidas para se criar, destruir e modificar registros, as instruções CONNECT, DISCONNECT e RECONNECT são fornecidas para criar, destruir e modificar “membros de conjuntos” – isto é, conexões entre um registro membro e um registro dono. Para efeito dos nossos exemplos, vamos introduzir um novo tipo de conjunto SETX (dono X, membro S, SET SELECTION IS BY APPLICATION; aqui S é o tipo usual de registro de fornecedor, e X é um novo tipo arbitrário de registro).

- Conecte a ocorrência S do fornecedor S4 à ocorrência SETX cujo dono é a ocorrência x de X.

O processo de conexão consiste essencialmente de (1) localizar a ocorrência requerida de SETX, (2) localizar a ocorrência requerida de S, e (3) executar a instrução CONNECT para conectar o último ao primeiro. O procedimento da etapa 1 pode ser qualquer um que torne a ocorrência de SETX a ocorrência *corrente* de SETX; tipicamente, um procedimento para encontrar (FIND) a ocorrência apropriada (x) de seu dono X.

1. FIND . . . x
2. FIND S occurrence for S4
3. CONNECT S TO SETX

A instrução CONNECT faz a conexão do corrente da unidade de corrida com uma ocorrência do conjunto especificado. A ocorrência envolvida é determinada pelo critério do SET SELECTION do conjunto especificado. Observe que a classe de membro de S em SETX não pode ser AUTOMATIC, a menos que seja também OPTIONAL (e a ocorrência S de S4 não pode ser correntemente um membro de qualquer ocorrência de SETX). Observe também no exemplo que a etapa 1 *tem* que ser executada antes da etapa 2 (Por quê?). Após o CONNECT, a ocorrência S de S4 é o registro corrente do conjunto SETX.

26.9 DISCONNECT

- Desconecte a ocorrência S de S4 da ocorrência SETX que a contém. (Veja o exemplo na Seção 26.8.)

```
FIND S occurrence for S4  
DISCONNECT S FROM SETX
```

A instrução DISCONNECT desconecta o corrente da unidade de corrida da ocorrência do conjunto especificado que a contém; a ocorrência do registro ainda existe no banco de dados, mas não é mais um membro do conjunto especificado. Se o corrente da unidade de corrida era previamente o registro corrente do conjunto especificado, então o indicador de corrente daquele conjunto é ajustado para ainda identificar a posição (o “buraco”) no conjunto, previamente ocupada pelo registro agora desconectado, mas *não* identifica aquele registro. Observe que S tem que ser um membro OPTIONAL de SETX; observe também que a ocorrência S de S4 tem que ser correntemente um membro de alguma ocorrência SETX.

26.10 RECONNECT

- Desconecte a ocorrência S da ocorrência SETX que a contém e conecte-a à ocorrência SETX cujo dono é a ocorrência x de X. (Veja exemplos nas Seções 26.8 e 26.9.)

1. FIND . . . x
2. FIND S occurrence for S4
3. RECONNECT S WITHIN SETX

A instrução RECONNECT desconecta o corrente da unidade de corrida da ocorrência do conjunto especificado que a contém, e conecta-a a alguma ocorrência — possivelmente a mesma — daquele conjunto. Esta última ocorrência é determinada pelo critério de SET SELECTION do conjunto especificado. O efeito sobre os indicadores do corrente é exatamente como para DISCONNECT, *exceto* que, se durante o estágio de conexão da operação o “buraco” no qual o registro for conectado estiver identificado pelo indicador de corrente do conjunto, o indicador do corrente será ajustado para identificar o registro reconectado. Observe no exemplo que S não pode ser um membro FIXED de SETX a menos que a ocorrência de SETX da qual o registro é desconectado e a ocorrência de SETX na qual o registro é reconectado sejam uma e a mesma.³ Observe também que (como na Seção 26.8), as etapas 1 e 2 não podem ser intercambiadas.

26.11 FIND

O formato básico da instrução FIND é:

FIND expressão-de-seleção-do-registro

onde “expressão-de-seleção-do-registro” (abreviadamente r-s-e) é uma expressão que designa alguma ocorrência de registro no banco de dados. A função de FIND é localizar a ocorrência designada e torná-la o corrente da unidade de corrida — e também o registro cor-

³ Algumas vezes é necessário movimentar-se registros dentro de uma mesma ocorrência de conjunto, ao invés de movimentá-los de uma ocorrência de conjunto para outra. Em particular, esta função pode ser requerida se o conjunto envolvido for singular. O RECONNECT permite tais movimentações, mesmo que o membro seja FIXED (quando DISCONNECT e CONNECT não poderiam ser usados).

rente de todos os realms e todos os conjuntos nos quais ele participe, e o registro corrente do seu tipo de registro (mas veja a Seção 26.11.7). Há seis formatos genéricos de r-s-e, e portanto seis formatos de FIND. (Originalmente havia sete, mas o formato 1 foi descontinuado. Renumeramos os formatos 2–7 como 1–6 na matéria que se segue. De qualquer maneira, escolhemos não discuti-los na ordem numérica, por razões didáticas.)

26.11.1 Acesso dentro de um tipo de registro (Formato 1) – O formato 1 básico de FIND já foi ilustrado várias vezes. Ele fornece acesso direto a uma ocorrência de registro por meio do valor de algum item de dado (ou combinação de itens de dados) daquela ocorrência. Entretanto, há uma variante importante do formato 1 básico, que é necessário quando o item de dado envolvido é não-único (isto é, não foi especificado DUPLICATES NOT ALLOWED). Por exemplo, para encontrar todas as ocorrências S em que o valor de CITY é LONDON:⁴

```
MOVE 'LONDON' TO CITY IN S
FIND ANY S USING CITY IN S
MOVE 'NO' TO NOTFOUND
PERFORM UNTIL NOTFOUND = 'YES'
    GET S
    .
    .
    .
    FIND DUPLICATE S USING CITY IN S
END-PERFORM
```

O efeito desta codificação é o seguinte. As duas primeiras instruções localizam *alguma* ocorrência de S cujo valor de city é LONDON (supondo que exista uma; qual é particularmente encontrada depende da definição do implementador). A terceira instrução inicializa com 'NO' o item de dado NOTFOUND. Depois, o *loop* de PERFORM recupera a ocorrência S recém-encontrada, processa-a, e procura encontrar uma "duplicata" daquela ocorrência S. Duplicata aqui é uma ocorrência S tendo o mesmo valor de CITY do registro corrente do tipo S. O *loop* é repetido até que NOTFOUND seja ajustado para 'YES' (o que deve ocorrer como resultante de um procedimento declarativo de "registro não encontrado", como na Seção 26.3). A codificação garantidamente encontrará todas as ocorrências requeridas de S, e não encontrará qualquer ocorrência duas vezes, embora a seqüência precisa na qual os registros são encontrados dependa da definição do implementador.

A cláusula USING pode tomar a forma "USING item-de-dado, item-de-dado, . . .", caso em que a busca é de alguma ocorrência que contenha os valores desejados dos itens de dados especificados. Pode ser útil pensar-se no acesso via FIND formato 1 como uma abstração da randomização.

26.11.2 Acesso ao dono (Formato 5) – Suponha que o registro corrente de um conjunto P-SP é uma ocorrência particular de SP. Encontre a ocorrência P correspondente.

```
FIND OWNER WITHIN P-SP
```

4

Neste exemplo usamos a construção PERFORM/END-PERFORM introduzida como parte da extensão de programação estruturada do COBOL em [23.3]. Futuros exemplos usarão também a construção IF/END-IF, da mesma fonte.

Em geral, "FIND OWNER" encontra o dono no tipo especificado de conjunto da ocorrência corrente de conjunto daquele tipo. Observe que o registro corrente do conjunto pode realmente já ser o dono.

26.11.3 Acesso seqüencial dentro de um conjunto ou realm (Formato 3) – Encontre os valores de PNO das peças fornecidas pelo fornecedor S4.

```
MOVE 'S4' TO SNO IN S
FIND ANY S USING SNO IN S
MOVE 'NO' TO EOF
FIND FIRST SP WITHIN S-SP
PERFORM UNTIL EOF = 'YES'
    GET SP
    (add PNO IN SP to result list)
    FIND NEXT SP WITHIN S-SP
END-PERFORM
```

Estão ilustradas neste exemplo duas versões do FIND formato 3. O FIND FIRST localiza a primeira ocorrência SP dentro da ocorrência corrente do conjunto S-SP (a ocorrência S-SP cujo dono é o fornecedor S4). Depois, a cada interação do *loop*, o FIND NEXT localiza a próxima ocorrência SP dentro da ocorrência corrente do conjunto S-SP (a ocorrência S-SP que contém a ocorrência SP mais recentemente encontrada), relativa à posição identificada pelo indicador do corrente S-SP (a posição da ocorrência mais recentemente encontrada). Estamos supondo que o item de dado EOF será ajustado para 'YES' quando não houver próxima ocorrência SP a ser encontrada.

A especificação de NEXT (ou FIRST) no FIND formato 3 pode ser substituída por PRIOR, LAST, um inteiro *n*, ou o nome de um item de dado contendo um valor inteiro. Nos dois últimos casos, um inteiro positivo representa o número da ocorrência desejada do registro contado na direção do NEXT a partir do início da ocorrência do conjunto (1 = FIRST), e um inteiro negativo representa o número da ocorrência do registro desejado contado na direção PRIOR a partir do final da ocorrência do conjunto (-1 = LAST). Além disso, o nome de conjunto pode ser substituído por um nome de realm, caso em que NEXT, PRIOR, etc. são interpretados como posições dentro do realm indicado. A seqüência dos registros dentro de um realm é definida pelo implementador.

26.11.4 Acesso seqüencial dentro de conjunto (Formato 6) – Encontre a quantidade de peças P5 fornecidas pelo fornecedor S1.

```
MOVE 'S1' TO SNO IN S
FIND ANY S USING SNO IN S
MOVE 'P5' TO PNO IN SP
FIND SP WITHIN S-SP CURRENT USING PNO IN SP
GET SP
(Print QTY IN SP)
```

A quarta instrução aqui é um FIND formato 6. Ela opera localizando a primeira ocorrência do registro SP dentro da ocorrência corrente do conjunto S-SP que tem um valor de PNO igual ao valor de PNO IN SP na UWA (isto é, um valor de PNO igual a 'P5').

Observe, incidentalmente, que estamos face a um problema de estratégia neste exemplo: ao invés de começar com o fornecedor e pesquisar a ocorrência S-SP correspon-

dente, poderíamos ter começado com a peça e pesquisado a ocorrência P-SP correspondente.⁵ Se for omitido o CURRENT de um FIND formato 6, a ocorrência de conjunto a ser pesquisada fica determinada de acordo com a especificação de SET SELECTION para o conjunto envolvido. Portanto, no procedimento acima poderia ter sido omitida a instrução FIND ANY S USING SNO IN S, pois o CURRENT havia sido omitido também no outro FIND. Há não apenas dois, mas quatro procedimentos possíveis de serem usados para responder à questão original (começar com S1 ou P5, incluindo ou omitindo CURRENT). Como exercício, o leitor poderia considerar o efeito sobre o posicionamento corrente em cada um dos casos (naturalmente, cada um deles resultará no mesmo corrente da unidade de corrida, mas outros valores de corrente *não* serão os mesmos em cada caso).

26.11.5 Acesso seqüencial dentro de conjunto (Formato 2) – Encontre todos os embarques do fornecedor S1 em que a quantidade seja 100.

```
MOVE 'S1' TO SNO IN S
FIND ANY S USING SNO IN S
MOVE 100 TO QTY IN SP
FIND SP WITHIN S-SP CURRENT USING QTY IN SP
MOVE 'NO' TO NOTFOUND
PERFORM UNTIL NOTFOUND = 'YES'
    GET SP
    ...
    FIND DUPLICATE WITHIN S-SP USING QTY IN SP
END-PERFORM
```

As primeiras quatro instruções localizam a primeira ocorrência SP do fornecedor S1 que tem um valor de QTY de 100 (isto é, a ocorrência SP ‘S1/P5/100’; O procedimento é essencialmente o mesmo do exemplo anterior e usa o FIND formato 6). A próxima instrução inicializa NOTFOUND com ‘NO’. Dentro do *loop* de PERFORM, o FIND formato 2 (FIND DUPLICATE ...) pesquisa a ocorrência corrente S-SP na direção NEXT, começando no registro corrente do conjunto S-SP e buscando a próxima ocorrência SP que tenha o mesmo valor de QTY do registro corrente. Observe que o FIND formato 6 não pode ser usado neste ponto, pois ele simplesmente localizaria ‘S1/P5/100’ novamente; precisamos de uma instrução que pesquise para frente da posição corrente, como faz o FIND formato 2. O *loop* é repetido até que ocorra uma condição de “não encontrado”.

É importante entender que o efeito da cláusula ‘USING QTY IN SP’ no formato 2 do FIND é fazer com que o DBMS pesquise a próxima ocorrência com o mesmo valor de QTY do registro corrente – não o mesmo valor do item de dado QTY IN SP na UWA. (Contrasta com a semântica da mesma cláusula no FIND formato 6.) No exemplo acima, os dois valores são de fato o mesmo, mas a instrução “GET SP” poderia ter sido substituída pela instrução, digamos

```
MOVE 400 TO QTY IN SP
```

sem afetar em nada qual ocorrência SP seria encontrada pelo FIND formato 2.

Caso o banco de dados inclua um conjunto singular ligando todas as ocorrências SP, haverá uma terceira possibilidade.

A cláusula USING nas instruções FIND formato 2 e formato 6 pode tomar a forma “USING item-de-dado, item-de-dado, . . .”, caso em que a pesquisa seria por uma ocorrência contendo os valores desejados de todos os itens de dados especificados.

26.11.6 Acesso por chave do banco de dados (Formato 4) — O FIND formato 4 é usado para se “encontrar” o registro que tenha um valor especificado de chave do banco de dados. Colocamos “encontrar” entre apóstrofos para enfatizar o fato de que, em um certo sentido, o registro envolvido já deverá ter sido encontrado, pois em última análise a única forma pela qual a unidade de corrida pode descobrir um valor de chave de um registro do banco de dados é tornando aquele registro o corrente da unidade de corrida. O formato 4 difere dos outros formatos FIND por ser sua *única* função a de atualizar a tabela de indicadores de corrente; ele não requer qualquer acesso ao banco de dados.

A “expressão-de-seleção-do-registro” no FIND formato 4 toma a forma de um *identificador de chave do banco de dados*. Por sua vez, um identificador de chave do banco de dados pode ter duas formas; discutiremos uma de cada vez.

Caso 1

Estabelece o registro corrente do conjunto S-SP como o corrente da unidade de corrida. (Observe que este registro não é necessariamente o corrente da unidade de corrida.) Vamos supor que o registro corrente do conjunto S-SP seja uma ocorrência SP.

FIND CURRENT SP WITHIN S-SP

Se um nome de registro e um “WITHIN conjunto” forem ambos especificados, como no exemplo, então o registro corrente do conjunto indicado tem que ser do tipo indicado. A referência do WITHIN pode ser a um realm ao invés de a um conjunto, caso em que o registro a ser encontrado é o registro corrente daquele realm; novamente, o registro corrente tem que ser do tipo indicado se for especificado um nome de registro. Se o WITHIN for omitido mas o nome do registro for especificado, então o registro a ser encontrado será o registro corrente do tipo indicado de registro. Se forem omitidos o nome do registro e o WITHIN, o registro a ser encontrado será o corrente da unidade de corrida. Portanto, a forma geral da “chave identificadora do banco de dados” (caso 1) é:

CURRENT [nome de registro] [WITHIN nome]

onde “nome” pode ser um nome de conjunto ou um nome de realm.

Caso 2

o segundo caso baseia-se na noção de *keep lists*. Keep lists estão discutidas em detalhes na Seção 26.12, e portanto deixaremos a discussão detalhada sobre este formato FIND para a mesma seção. Entretanto, para que o trabalho esteja completo, apresentamos a sintaxe do segundo caso de “chave identificadora do banco de dados” aqui:

posição WITHIN nome-da-keep-List

onde “posição” é FIRST ou LAST.

Exemplo:

FIND LAST WITHIN LISTA

26.11.7 O RETAINING – Para cada fornecedor que fornece a peça P4, encontre outra peça fornecida pelo mesmo fornecedor, e imprima o número do fornecedor, o nome do fornecedor e o número da peça. Por simplicidade, suponha que cada fornecedor que fornece a peça P4 também fornece pelo menos uma outra peça.

Em esboço, o procedimento requerido é o seguinte. Começando pela peça P4, inspecionamos cada ocorrência SP daquela peça. Para cada ocorrência SP, extraímos SNO e SNAME da ocorrência S correspondente, e então pesquisamos as ocorrências SP daquele fornecedor, procurando uma que ligue o fornecedor a uma peça que *não* seja P4. Assim que encontrarmos essa ocorrência SP, extraímos o PNO relevante. Assim, com base nos dados de amostragem da Fig. 26.1, um resultado possível desse procedimento seria

SNO	SNAME	PNO
S1	Smith	P1
S4	Clark	P2

A seguir apresentamos uma primeira aproximação à codificação deste problema usando a DML COBOL.

```
1 MOVE 'P4' TO PNO IN P
2 FIND ANY P USING PNO IN P
3 MOVE 'NO' TO EOF
4 PERFORM UNTIL EOF = 'YES'
5   FIND NEXT SP WITHIN P-SP
6   IF EOF NOT = 'YES'
7     FIND OWNER WITHIN S-SP
8     GET S
9     MOVE 'NO' TO FOUND
10    PERFORM UNTIL FOUND = 'YES'
11      FIND NEXT SP WITHIN S-SP
12      GET SP
13      IF PNO IN SP NOT = 'P4'
14        MOVE 'YES' TO FOUND
15      END-IF
16    END-PERFORM
17    (print SNO IN S, SNAME IN S, PNO IN SP)
18  END-IF
19 END-PERFORM
```

(Com base na nossa simplificação, não testamos a condição de final de conjunto no *loop* interno. Observe que FIND NEXT é equivalente a FIND FIRST quando o registro corrente de um conjunto é um dono.)

A codificação não está, entretanto, correta; ela contém um erro lógico. Tente en-

contrar o erro antes de ler a explicação abaixo; talvez seja uma boa idéia “executar” o procedimento com os dados de amostragem da Fig. 26.1. O resultado é

SNO	SNAME	PNO
S1	Smith	P1
S2	Jones	P1

É interessante observar que o procedimento não termina de forma anormal em nenhuma situação. Ele simplesmente dá uma resposta que parece certa, mas que de fato está errada.

O erro é o seguinte. Quando é executado, o FIND da linha 11 estabelece uma ocorrência SP como o corrente da unidade de corrida. Esta ocorrência, sendo a ocorrência SP que recebeu o acesso mais recente, torna-se também o registro corrente de todos os conjuntos nos quais participa – incluindo, em particular, o conjunto P-SP. Este, por sua vez, torna corrente a ocorrência P-SP que contém esta ocorrência SP. Assim, quando é executado o FIND da linha 5, na próxima interação do *loop* externo (procurando o próximo fornecedor de P4), a ocorrência P-SP referenciada por aquela instrução não será (em geral) aquela cujo dono é P4. Para evitar esta situação, o FIND da linha 11 tem que ser estendido para incluir um RETAINING apropriado:

```
FIND NEXT SP WITHIN S-SP  
RETAINING P-SP CURRENCY
```

O efeito desta colocação é o de evitar a atualização do indicador do corrente para o conjunto P-SP. Em geral, pode ser suprimida a atualização do posicionamento corrente para todos os realms envolvidos,⁶ para o tipo de registro envolvido, para qualquer ou todos os tipos de conjuntos envolvidos, ou qualquer combinação destes; veja em [23.3] os detalhes de sintaxe em cada caso. O único indicador do corrente que nunca pode ser suprimido é o “corrente da unidade de corrida”.

O RETAINING pode ser especificado em qualquer instrução FIND ou STORE.

26.12 LISTAS DE RETENÇÃO (KEEP LISTS)

- Seja M o valor máximo de status de todos os fornecedores de Paris. Coloque M como valor de status para todos os fornecedores de Paris.

É claro que este problema requer duas interações sobre o conjunto dos fornecedores de Paris – uma para se determinar o valor M e uma para executar as atualizações. (O termo conjunto está sendo usado aqui no seu sentido usual, não no sentido especial DBTG.) Segue-se um procedimento possível.

⁶ Lembre-se de que um determinado registro pode pertencer a múltiplos realms. Não é possível suprimir-se a atualização do corrente para alguns realms sem se fazer o mesmo para outros.

```

MOVE 'PARIS' TO CITY IN S
MOVE ZERO TO M
MOVE 'NO' TO NOTFOUND
FIND ANY S USING CITY IN S
PERFORM UNTIL NOTFOUND = 'YES'
    GET STATUS IN S
    IF STATUS IN S > M
        MOVE STATUS IN S TO M
    END-IF
    FIND DUPLICATE S USING CITY IN S
END-PERFORM

MOVE M TO STATUS IN S
MOVE 'NO' TO NOTFOUND
FIND ANY S USING CITY IN S
PERFORM UNTIL NOTFOUND = 'YES'
    MODIFY STATUS IN S
    FIND DUPLICATE S USING CITY IN S
END-PERFORM

```

Entretanto, como ambos os *loops* interagem exatamente sobre o mesmo conjunto de registros, seria mais eficiente se, de alguma forma, pudéssemos anotar os endereços (valores das chaves do banco de dados) dos registros encontrados no primeiro *loop*, indo diretamente a esses registros no segundo *loop* via seus endereços, ao invés de pesquisá-los novamente a partir da estaca zero. As *listas de retenção* fornecem uma base para essa abordagem. Uma lista de retenção é um objeto com nome cuja função é manter uma lista ordenada de valores de chaves do banco de dados. Uma unidade de corrida pode ter qualquer quantidade de listas de retenção; observe que as listas de retenção não fazem parte do banco de dados, sendo locais à unidade de corrida que as usa. São definidas na Data Division do programa, seguindo-se à instrução que “invoca” o subesquema. Por exemplo,

```

DB SUPPLIERS WITHIN SUPPLIERS-AND-PARTS.
LD LISTA LIMIT IS 15.
LD LISTB LIMIT IS 20.

```

Durante a execução deste programa, as listas de retenção LISTA e LISTB conterão cada uma uma lista ordenada dos valores de chaves do banco de dados, designando assim um conjunto de registros no banco de dados. LISTA pode manter até 15 desses valores, e LISTB até 20. Os valores de chaves do banco de dados podem ser adicionados a uma lista de retenção via instrução KEEP, e removidos via uma instrução FREE. Essas instruções serão discutidas mais tarde em detalhes.

Voltando ao nosso problema, mostraremos uma solução usando uma lista de retenção (LISTA, como definido acima; vamos supor que não haja mais do que 15 fornecedores em Paris). Cada vez que é executada uma instrução KEEP, ela adiciona o valor da chave do banco de dados do corrente da unidade de corrida (um fornecedor de Paris, no nosso exemplo) ao final da lista de retenção especificada (LISTA). Portanto, ao sair do primeiro *loop*, LISTA contém os valores de chaves do banco de dados de todos os fornecedores de Paris, na ordem em que eles foram encontrados.

```

MOVE 'PARIS' TO CITY IN S
MOVE ZERO TO M
MOVE 'NO' TO NOTFOUND
FIND ANY S USING CITY IN S
PERFORM UNTIL NOTFOUND = 'YES'
    KEEP USING LISTA
    GET STATUS IN S
    IF STATUS IN S > M
        MOVE STATUS IN S TO M
    END-IF
    FIND DUPLICATE S USING CITY IN S
END-PERFORM

```

```

MOVE M TO STATUS IN S
MOVE 'NO' TO LISTEMPTY
PERFORM UNTIL LISTEMPTY = 'YES'
    FIND FIRST WITHIN LISTA
    FREE FIRST WITHIN LISTA
    MODIFY STATUS IN S
END-PERFORM

```

O segundo *loop* então interage com os registros indicados por esses valores de chaves do banco de dados; em cada interação, “encontra” o registro indicado pelo primeiro valor da lista, e remove esse valor da lista. (Observe, entretanto, que a condição de término do *loop* não é “registro não-encontrado”, mas “lista de retenção vazia”. Não mostraremos detalhes de como manusear esta condição, mas suporemos simplesmente que segue o padrão usual.) Como mencionamos na Seção 26.11.6, a expressão FIRST WITHIN LISTA é uma “chave identificadora do banco de dados”.

A forma geral de KEEP é

KEEP [chave-identificadora-do-banco-de-dados] USING nome-da-lista-de-retenção.

onde os formatos possíveis para a “chave identificadora do banco de dados” são

CURRENT [nome-do-registro] [WITHIN {^{nome-de-conjunto}_{nome-de-realm}}]

e

{FIRST {LAST {WITHIN nome-da-lista-de-retenção}}

(As opções colocadas verticalmente entre chaves mostram que elas são alternativas.) O efeito da instrução KEEP é o seguinte. O valor de chave do banco de dados representado pelo identificador da chave do banco de dados – ou, se não for especificado esse identificador, o valor da chave do banco de dados do corrente da unidade de corrida – é adicionado ao final da lista de retenção especificada. A forma CURRENT do identificador da chave do banco de dados é como explicado na Seção 26.11.6; as formas FIRST/LAST são auto-explicativas. Um determinado valor de chave do banco de dados pode aparecer em diversas listas de retenção distintas, ou diversas vezes na mesma lista de retenção, ou ambos.

O formato geral de FREE é

FREE identificador-da-chave-do-banco-de-dados

ou

FREE ALL [FROM nome-da-lista-de-retenção]

No primeiro formato, se o identificador da chave do banco de dados for uma das formas CURRENT, o indicador do corrente especificado é ajustado para nulo; se for uma das formas da lista de retenção (FIRST ou LAST), o valor indicado é removido da lista de retenção indicada. No segundo formato, todos os valores da lista de retenção indicada (ou, se FROM for omitido, de todas as listas de retenção) são removidos; em outras palavras, a lista de retenção indicada fica vazia (ou, se FROM for omitido, todas as listas de retenção ficam vazias).

26.13 INSTRUÇÕES DIVERSAS

Concluiremos este capítulo com uma breve descrição das restantes instruções DML (READY e FINISH, COMMIT e ROLLBACK.)

26.13.1 READY e FINISH – A instrução READY torna um realm disponível para processamento – por exemplo,

READY S-SP-P USAGE-MODE IS UPDATE

USAGE-MODE pode ser RETRIEVAL ou UPDATE. Cada um destes pode ser adicionalmente qualificado como PROTECTED ou EXCLUSIVE, indicando a exigência da unidade de corrida no tocante ao compartilhamento do realm com as unidades de corrida concorrentes; PROTECTED significa que o realm pode ser compartilhado com outras recuperações mas não atualizações, e EXCLUSIVE significa que não pode ter qualquer compartilhamento.

FINISH torna o realm que estava disponível não mais disponível – por exemplo,

FINISH S-SP-P

26.13.2 COMMIT e ROLLBACK – Estas duas instruções (que não têm outros operandos) são as análogas, no DBTG, das operações DL/I SYNC e ROLB (Capítulo 22). Vamos discuti-las aqui muito superficialmente.

A instrução COMMIT estabelece um “ponto de parada” (synchpoint) para a unidade de corrida. (O início da unidade de corrida também é considerado um ponto de parada.) Todas as alterações feitas no banco de dados pela unidade de corrida desde o ponto de parada anterior são “committed” – isto é, são tornadas visíveis às unidades de corrida concorrentes, e garantidas de que não serão mais desfeitas (veja “rollback” abaixo). A última instrução executada normalmente em uma unidade de corrida deve ser COMMIT, pois ao final da unidade de corrida o DBMS automaticamente desfaz qualquer alteração ao banco de dados feita pela unidade de corrida desde o último ponto de parada. (Este é freqüentemente, mas não invariavelmente, o único COMMIT da unidade de corrida).

A instrução ROLLBACK desfaz todas as modificações feitas pela unidade de corrida no banco de dados desde o último ponto de parada. Como mencionamos acima, o DBMS emite automaticamente um ROLLBACK ao término da unidade de corrida. A unidade de corrida pode decidir emitir um ROLLBACK se, por exemplo, descobrir algum erro no meio do processamento de uma entrada *batch*. Observe, entretanto, que a unidade de corrida continua seu processamento após a execução do ROLLBACK, e que portanto pode fazer outras atualizações que não serão desfeitas.

EXERCÍCIOS

26.1 Usando a sua resposta ao Exercício 24.3 (esquema para o banco de dados de fornecedores-peças-projetos) como base, forneça soluções DBTG às seguintes questões (Exercícios 26.1.1-26.1.9).

- 26.1.1 Obtenha valores SNO dos fornecedores que fornecem para o projeto J1.
 - 26.1.2 Obtenha valores SNO dos fornecedores que fornecem a peça P1 ao projeto J1.
 - 26.1.3 Obtenha valores SNO dos fornecedores que fornecem uma peça vermelha para o projeto J1.
 - 26.1.4 Obtenha valores PNO das peças fornecidas para todos os projetos de Londres.
 - 26.1.5 Obtenha valores JNO dos projetos que não recebem nenhuma peça vermelha de nenhum fornecedor de Londres.
 - 26.1.6 Obtenha valores PNO das peças fornecidas por pelo menos um fornecedor que fornece pelo menos uma peça suprida pelo fornecedor S1.
 - 26.1.7 Obtenha todos os pares de valores CITY tais que um fornecedor na primeira cidade forneça para um projeto na segunda cidade.
 - 26.1.8 Mude a cor de todas as peças vermelhas para laranja.
 - 26.1.9 A quantidade de P1 fornecida por S1 a J1 deve passar a ser fornecida por S2. Faça todas as alterações necessárias.
- 26.2 Suponhamos que o registro SPJ no esquema de fornecedores-peças-projetos contenha somente o item de dado QTY; isto é, foi eliminada a gravação redundante de SNO, PNO, e JNO. Ainda é possível – por procedimento – manter a ordenação desejada das ocorrências SPJ dentro de qualquer ocorrência de conjunto – por exemplo, manter todas as ocorrências de um determinado fornecedor em ordem de número de projeto dentro de uma ordem de número de peça. (Observe, entretanto, que esta mudança terá repercussões consideráveis nos programas que lidam com a manutenção desses conjuntos.) Suponhamos, portanto, que a redundância foi eliminada mas que a ordenação dentro dos conjuntos foi mantida. Que efeito trará isto sobre as suas soluções do Exercício 26.1? (Nota: será necessária uma consulta a [23.3] e [23.4] para o último destes, para saber como manter a ordenação dentro de um conjunto sem que tenha sido especificado ORDER IS SORTED.)
- 26.3 Considere o esquema de “estrutura gerencial” do Exercício 24.2. Escreva instruções para criar uma ocorrência EMP para o empregado E15, e para colocá-lo no ponto apropriado da hierarquia. Pode assumir que as ocorrências EMP para os empregados E1 e E8 (veja Fig. 24.7) já existem, e que também existe uma ocorrência LINK conectando-os; não assuma, entretanto, que já exista uma ocorrência LINK subordinada a E8. Estabeleça suas hipóteses com relação à classe de membro e SET SELECTION.
- 26.4 Considere o esquema “peças e componentes” do Exercício 24.4.
- 26.4.1 Para cada peça, imprima o número da peça, os números de seus componentes imediatos e os números das peças das quais esta é um componente imediato (montagens imediatas).
 - 26.4.2 Para uma determinada peça, imprima os números de todas as suas peças componentes, em todos os níveis (problema de explosão das peças).

Assuma que o esquema inclui um conjunto singular chamado PART-SET, que contém como membros todas as ocorrências PART.

REFERÊNCIAS E BIBLIOGRAFIA

- 26.1 C. W. Bachman. "The Programmer as Navigator." *CACM* 16, nº 11 (novembro de 1973).

Contém a apresentação feita por Bachman por ocasião do recebimento do Prêmio Turing de 1973. Bachman faz um contraste entre o ponto de vista antigo sobre processamento de dados, em que o computador era o centro e os dados considerados como fluindo pela máquina à medida que eram processados, com o ponto de vista mais moderno, no qual o banco de dados é o recurso principal e o computador somente uma ferramenta de acesso. O termo "navegação" é usado para descrever o processo de se conduzir através do banco de dados, seguindo caminhos explícitos de um registro para o próximo, na busca de uma parte determinada de dado.

Parte 5

Revisão

das Três

Abordagens

O objetivo desta parte do livro é o de reunir alguns dos temas introduzidos nos capítulos precedentes, e analisar alguns aspectos das três abordagens com maior profundidade do que antes. O Capítulo 27 introduz a Linguagem Unificada de Bancos de Dados (UDL), uma linguagem para programação de aplicações que suporta as três abordagens e, com isso, proporciona uma base conveniente para compará-las ao nível externo. O Capítulo 28 apresenta uma análise detalhada e uma comparação entre os méritos relativos de relações e redes, como bases para o nível conceitual. Não estamos nos propondo, entretanto, que o tratamento dado neste material seja exaustivo.

27

A Linguagem Unificada de Banco de Dados

27.1 INTRODUÇÃO

Como já mencionamos, a Linguagem Unificada de Bancos de Dados (UDL) é uma linguagem que suporta as três abordagens bem conhecidas — relações, hierarquias e redes — de uma maneira uniforme e consistente. Ela não é uma linguagem autocontida; ao invés, é feita para ser explicitamente “fortemente acoplada” como extensão a linguagens de programação existentes (COBOL, PL/I, . . .), e pode ser incorporada, com modificações de sintaxe adequadas, a uma variedade de linguagens principais. Além de suportar as três abordagens, a UDL também proporciona operações de um registro por vez e um conjunto por vez a cada uma delas, como veremos. (O nível de conjunto é desejável por razões de produtividade e facilidade de programação; o nível de registro serve como ponte à função existente na linguagem principal. Além disso, há alguns problemas para os quais o nível de registro é de qualquer forma mais adequado [veja, por exemplo, o Exercício 27.2, segunda parte].)

Uma introdução geral à UDL encontra-se em [27.1]; especificações detalhadas das versões para COBOL e PL/I podem ser encontradas em [27.3–27.6]. Usaremos a versão PL/I como base para a maior parte das nossas discussões. À época da confecção deste livro, não havia implementação disponível da UDL.

27.2 A ABORDAGEM DA CONJUNÇÃO

A UDL destina-se não somente a suportar as três abordagens, mas a fazê-lo conjugando o mais possível as linguagens através das diferentes estruturas. A observação a seguir é a chave para o alcance desse objetivo.

- Uma relação pode ser considerada como um caso especial de uma hierarquia — uma que seja “somente raiz”. Da mesma forma, uma hierarquia pode ser considerada como um caso especial de uma rede — uma em que cada registro filho tem exatamente um registro pai.

Reconhecendo esta formação entre as três estruturas de dados, a linguagem é capaz de manuseá-las de uma maneira unificada. Naturalmente, a “observação chave” é muito vaga para ter outra utilidade que não a de um sentido geral de direção. Podemos torná-la mais precisa a seguir.

1. Definimos um *conjunto de registros* como um conjunto de pares ordenados $\langle R_i, P_i \rangle$ tais que todos os registros R_i são de um mesmo tipo, e não há dois registros com a mesma posição (endereço) P_i .
2. O conjunto particular de registros que envolve *todos* os registros de um determinado tipo (em um determinado banco de dados) é o *conjunto base* daquele tipo de registro.
3. Um *banco de dados* é uma coleção de um ou mais conjuntos base, juntamente com zero ou mais conjuntos agregados (veja o nº 7 abaixo).¹
4. Uma *relação* é um conjunto de registros para os quais $i \neq j$ (veja o nº 1 acima) implica $R_i \neq R_j$.
5. Um *banco de dados relacional* é uma coleção de um ou mais conjuntos de base, cada um dos quais é uma relação.
6. Um *agregado* dos conjuntos de base P e C , nessa ordem, é um par ordenado $\langle \text{pai}, \text{filho} \rangle$, tal que o “pai” é um registro de P , e o “filho” é um subconjunto (possivelmente vazio) dos registros de C .
7. Um *conjunto agregado* dos conjuntos de base P e C , nessa ordem, é um conjunto de agregados de P e C , nessa ordem. Cada registro de P é o pai de exatamente um agregado de F (veja o nº 6), e não há outros registros pai em F ; cada registro de C é um filho de no máximo um agregado de F , e não há outros registros filhos em F . (Conjuntos agregados correspondem aos tipos de conjuntos do DBTG. Os termos DBTG para pai e filho são, respectivamente, dono e membro.)
8. Um *banco de dados em rede* é simplesmente um banco de dados como definido no nº 3 acima.
9. Um *banco de dados hierárquico* é um banco de dados em rede que pode ser dividido em partições (*hierarquias*) como se segue: (a) Cada conjunto de base pertence a exatamente uma partição; (b) nenhum conjunto agregado ultrapassa uma partição; (c) se o conjunto agregado F é definido com os conjuntos de base P e C , nessa ordem, então cada registro de C é um filho em exatamente – não “no máximo” – um agregado de F ; (d) dentro de uma partição, cada conjunto de base exceto um – o “raiz” – é o conjunto filho de exatamente um conjunto agregado; (e) dentro de uma partição, há um caminho da raiz a cada não-raiz – onde um caminho do conjunto de base A para o conjunto de base B é uma seqüência de conjuntos agregados F_1, F_2, \dots, F_n tal que A é o conjunto pai de F_1 , o conjunto filho de F_1 é o conjunto pai de F_2, \dots , e o conjunto filho de F_n é B .²

¹ Pode também incluir “seqüências”, que correspondem aos conjuntos singulares do DBTG, mas não vamos discuti-las aqui por problemas de espaço.

² Um banco de dados IMS consiste de uma única partição, no sentido desta definição. Além disso, o IMS usa “segmento” para se referir ao que estamos chamando de registro.

Alertamos o leitor para o fato de que as definições acima ainda são bastante incompletas. Além disso, não cobrem a “seqüência hierárquica” do IMS, que é a ordenação total de todos os registros da hierarquia, nem os tipos de conjuntos DBTG que têm mais de um tipo de membro (filho). Esses dispositivos são considerados como de menor importância. (É verdade que a seqüência hierárquica tem hoje uma importância que não é menor em IMS, mas isto se deve em grande parte à forma como está definido o DL/I. Na maioria dos casos, poderia ser oferecida uma função equivalente que não se baseasse nessa seqüência, e não é difícil encontrar-se situações em que o conceito é positivamente um obstáculo.)

Aceitas essas restrições, podemos ver das definições que, como mencionado antes, uma relação é um caso especial de hierarquia, e uma hierarquia é um caso especial de rede. Estes fatos estão refletidos na estrutura da linguagem que estamos descrevendo. Assim, a linguagem toda tem uma estrutura em “camadas”, como ilustrado na Fig. 27.1. Mais especificamente:

- Os dispositivos da linguagem requeridos para se declarar um banco de dados relacional são um subconjunto dos requeridos para se declarar um banco de dados hierárquico, e estes, por sua vez, são um subconjunto dos requeridos para se declarar um banco de dados em rede;
- os operadores de linguagem requeridos para a manipulação de um banco de dados relacional são um subconjunto dos requeridos para a manipulação de um banco de dados hierárquico, e estes, por sua vez, são um subconjunto dos requeridos para a manipulação de um banco de dados em rede;
- (para um dado operador, conforme for aplicável) os operandos da linguagem requeridos para um banco de dados relacional são um subconjunto dos requeridos para um banco de dados hierárquico, e estes, por sua vez, são um subconjunto dos requeridos para um banco de dados em rede.

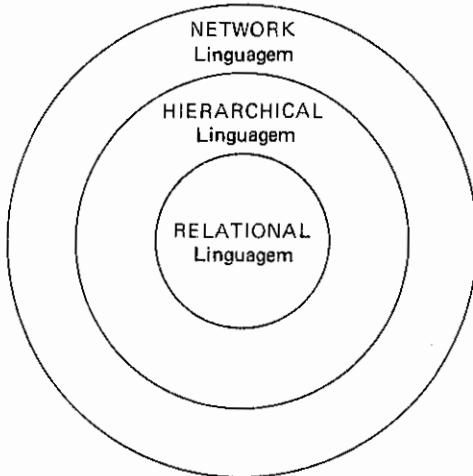


Fig. 27.1 A linguagem em camadas.

27.3 LINGUAGEM DECLARATIVA

Os dispositivos declarativos da UDL permitem uma declaração do banco de dados pela linguagem de programação – não confundir com a declaração de “sistema”, isto é, a declaração conhecida pelo DBMS básico. (A declaração de sistema é o esquema externo, em termos ANSI/SPARC.) Naturalmente as duas declarações não podem ser conflitantes; em algum ponto a declaração UDL tem que ser mapeada à declaração do sistema, mas detalhes deste processo fogem à estrutura da UDL em si.³ Entretanto, os programadores em UDL não terão normalmente que escrever as declarações UDL (no mínimo, deve ser possível obter essas declarações por COPY ou INCLUDE de alguma biblioteca fonte).⁴ Mas conceitualmente a declaração UDL faz parte do programa fonte; é usada pelo compilador ao compilar as instruções de manipulação da UDL, aparecerá na listagem do programa e tem que ser entendida pelo programador de UDL.

Apresentaremos a linguagem declarativa por meio de um exemplo (baseado em uma variante do banco de dados de educação do Capítulo 16). O banco de dados de educação contém informações sobre o esquema de treinamento interno de uma companhia. Para cada curso de treinamento, o banco de dados contém detalhes de todos os cursos pré-requisitos para aquele curso e de todas as ofertas daquele curso; e para cada oferta, contém detalhes de todos os professores e de todos os alunos daquela oferta. O banco de dados contém também informações sobre empregados. Uma estrutura relacional para estas informações está mostrada na Fig. 27.2; as Figs. 27.3 e 27.4 mostram, respectivamente, uma estrutura hierárquica e uma em rede para as mesmas informações.

COURSE	COURSE#	TITLE	
PREREQ	COURSE#	PRE#	
OFFERING	COURSE#	OFF#	DATE
TEACHER	COURSE#	OFF#	EMP#
STUDENT	COURSE#	OFF#	EMP#
EMPLOYEE	EMP#	NAME	

Fig. 27.2 Estrutura relacional do banco de dados de educação.

³ Como uma analogia, considere o caso de um arquivo convencional, para o qual existem: (a) uma declaração a nível de linguagem de programação [em qualquer programa fonte que se refere ao arquivo], (b) uma ‘declaração de sistema’ [consistindo, por exemplo, de entradas em um catálogo do sistema, e (c) um mapeamento entre essas duas [especificado por meio de instruções de job control].

⁴ Na prática, pode ser possível usar a declaração do sistema para gerar a declaração UDL (ou o oposto).

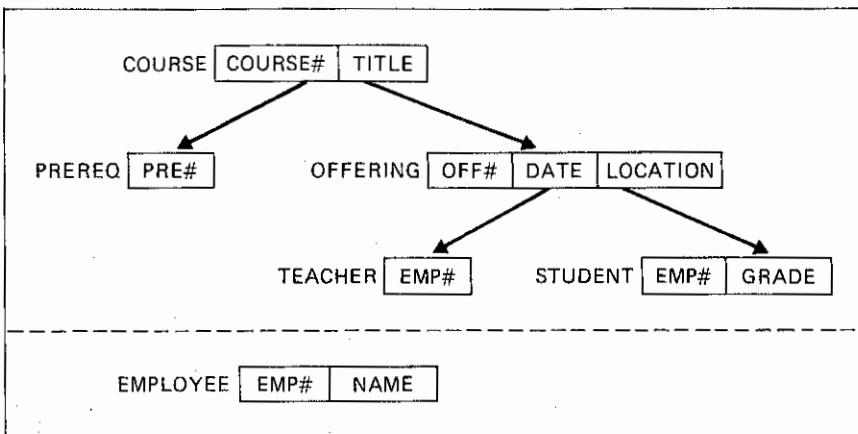


Fig. 27.3 Estrutura hierárquica do banco de dados de educação.

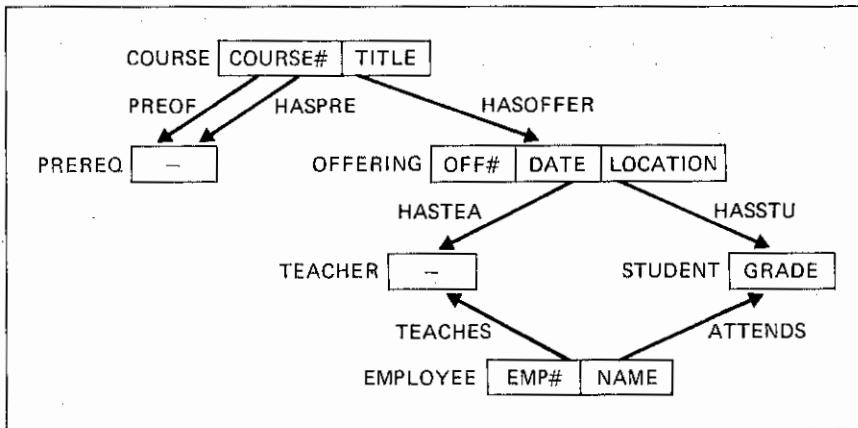


Fig. 27.4 Estrutura em rede do banco de dados de educação.

Observe que são necessárias duas hierarquias (uma delas “somente raiz”) na Fig. 27.3, para se evitar a redundância.

As declarações UDL correspondentes às Figs. 27.2, 27.3, e 27.4 definem um banco de dados contendo diversos *conjuntos de base* e *conjuntos agregados*. Consideremos primeiramente os conjuntos de base. (Para fins do exemplo, todos os conjuntos de base receberam nomes, embora na prática os nomes possam ser omitidos se não forem referenciados em nenhuma outra parte do programa. Estamos também supondo, por simplicidade, que todos os campos são do tipo de dado seqüência de caracteres (CHAR). Na prática, naturalmente, podem também ser suportados muitos outros tipos de dados.)

As linhas marcadas com um único asterisco são omitidas no caso de rede (Fig. 27.4); as linhas marcadas com um duplo asterisco são omitidas nos casos hierárquico e de rede (Figs. 27.3 e 27.4). (Essas omissões irão, naturalmente, levar a pequenos ajustes na pon-

tuação circundante.) As omissões são possíveis porque as informações relevantes são levadas pela estrutura de conjuntos agregados, ao invés de pôr chaves estrangeiras como no caso relacional.⁵ No caso da rede, as omissões fazem com que os tipos de registros PRE-REQ e TEACHER não contenham nenhum campo de dados (mas os tipos de registros ainda existem, e as ocorrências correspondentes ainda aparecem no banco de dados).

```
DCL EDUC DATABASE
BASESETS(
    CSET RECTYPE(1 COURSE BASED(C),
                 2 COURSE# CHAR(3),
                 2 TITLE CHAR(33))
                UNIQUE(COURSE#),
    PSET RECTYPE(1 PREREQ BASED(P),
                 2 COURSE# CHAR(3),
                 2 PRE# CHAR(3))
                UNIQUE,
    OSET RECTYPE(1 OFFERING BASED(O),
                 2 COURSE# CHAR(3),
                 2 OFF# CHAR(3),
                 2 DATE CHAR(6),
                 2 LOCATION CHAR(12))
                UNIQUE((COURSE#,OFF#)),
    TSET RECTYPE(1 TEACHER BASED(T),
                 2 COURSE# CHAR(3),
                 2 OFF# CHAR(3),
                 2 EMP# CHAR(6))
                UNIQUE,
    SSET RECTYPE(1 STUDENT BASED(S),
                 2 COURSE# CHAR(3),
                 2 OFF# CHAR(3),
                 2 EMP# CHAR(6),
                 2 GRADE CHAR(1))
                UNIQUE((COURSE#,EMP#)),
    ESET RECTYPE(1 EMPLOYEE BASED(E),
                 2 EMP# CHAR(6),
                 2 NAME CHAR(18))
                UNIQUE(EMP#))
```

Cada tipo de registro tem um *cursor* associado a ele (especificado por meio da cláusula BASED). Em UDL, um cursor é um “indicador de localização do banco de dados” – isto é, um indicador de localização que indica registros do banco de dados. C, por exemplo, é um cursor que será usado para indicar a localização dos cursos individuais (sómente). Também, C serve como o cursor *default* para referências implicitamente qualificadas a COURSE, na parte manipulativa do programa (veja a próxima seção). Podem ser defini-

5

Seria possível a inclusão das chaves estrangeiras nos casos hierárquico e de rede (por exemplo, OFFERINGS poderiam ainda incluir um campo COURSE#), mas então os conjuntos agregados tornar-se-iam “não-essenciais” – isto é, seriam redundantes do ponto de vista da informação. “Essencialidade” está discutida no Capítulo 28. Neste capítulo não consideraremos quaisquer conjuntos agregados não-essenciais.

dos cursos adicionais para um tipo de registro por meio de declaração explícita de cursor — por exemplo,

```
DCL C1 CURSOR RECTYPE(COURSE);
```

Cada cursor fica individualmente limitado a um único tipo de registro.

Para o caso relacional, cada conjunto de base tem uma cláusula UNIQUE, especificando que cada registro no conjunto de base tem um valor único no campo indicado ou em uma combinação de campos (em um determinado momento). A omissão da especificação do nome do campo na cláusula UNIQUE é equivalente à especificação da combinação de todos os campos do tipo de registro envolvido (veja, por exemplo, PREREQ). Para os casos hierárquico e de rede, UNIQUE está especificado para COURSE e EMPLOYEE somente — os outros quatro conjuntos de base não são relações nessas estruturas.

Qualquer um ou todos os conjuntos de base podem ser definidos como possuindo uma *ordenação*; por exemplo, podemos especificar ORDER (UP COURSE#) para cursos (e essa entrada implicaria UNIQUE (COURSE#), a menos que a entrada de ORDER também inclua a especificação NONUNIQUE). Para o caso relacional, as únicas ordenações válidas são controladas pelo valor e definidas pelo sistema; para as outras estruturas, podem ser especificados outros tipos de ordenação. Para maiores detalhes veja [27.4] e [27.5]. Por simplicidade, assumimos a ordenação *default* (definida pelo sistema) em todos os casos.

Para impor uma estrutura hierárquica ou de rede no banco de dados, a declaração tem que incluir especificações de FANSETS (conjuntos agregados), bem como de BASESETS (conjuntos de base) como já mostrado. Para a estrutura hierárquica da Fig. 27.3, esta poderia ser como se segue.

FANSETS

(RECORD(PREREQ)	UNDER(COURSE)	UNIQUE,
RECORD(OFFERING)	UNDER(COURSE)	UNIQUE(OFF#),
RECORD(TEACHER)	UNDER(OFFERING)	UNIQUE,
RECORD(STUDENT)	UNDER(OFFERING)	UNIQUE(EMP#))

Os quatro conjuntos agregados estão sem nome, embora não haja razão para não se dar um nome, se desejado. As entradas UNIQUE referem-se à unicidade dentro de cada agregado do conjunto agregado aplicável (por exemplo, cada oferta sob um determinado curso tem um número único de oferta). Também é possível especificar-se uma ORDER para o filho de cada agregado; novamente escolhemos, por simplicidade, a ordenação *default* (definida pelo sistema).

Para a estrutura em rede da Fig. 27.4, a entrada FANSETS poderia ser como a seguir. Desta vez os conjuntos agregados receberam nomes, embora possam permanecer sem nome se este não for requerido. Observe, entretanto, que os dois primeiros conjuntos agregados têm que ter nomes, pois de outra forma não seria possível fazer-se uma distinção entre eles (ambos consistem de PREREQs sob COURSES). Observe também que as cláusulas UNIQUE em muitos casos se referem a campos em *ancestrais* (pais, ou pais dos pais, ou . . .) do tipo de registro em questão. Poderiam também ser especificadas cláusulas UNIQUE no nível de conjuntos de base, se desejado.

FANSETS		UNDER(COURSE)
(PREOF	RECORD(PREREQ)	UNIQUE(COURSE# OVER PREREQ VIA HASPRE),
HASPRE	RECORD(PREREQ)	UNDER(COURSE) UNIQUE(COURSE# OVER PREREQ VIA PREOF),
HASOFFER	RECORD(OFFERING)	UNDER(COURSE) UNIQUE(OFF#), UNDER(OFFERING)
HASTEA	RECORD(TEACHER)	UNIQUE(EMP# OVER TEACHER), UNDER(OFFERING)
HASSTU	RECORD(STUDENT)	UNIQUE(EMP# OVER STUDENT), UNDER(EMPLOYEE)
TEACHES	RECORD(TEACHER)	UNIQUE((COURSE# OVER TEACHER, OFF# OVER TEACHER)), UNDER(EMPLOYEE)
ATTENDS	RECORD(STUDENT)	UNIQUE(COURSE# OVER STUDENT))

As entradas dos conjuntos agregados como estão mostradas ainda não estão completas. Em geral, para cada determinado conjunto agregado, o tipo de registro filho pode ser (a) AUTOMATIC ou MANUAL, e (b) FIXED, MANDATORY, ou OPTIONAL, em relação àquele conjunto agregado. (para uma hierarquia, somente são válidos AUTOMATIC e FIXED, e esses são os *defaults*.) Os significados desses termos são como para o DBTG.

27.4 LINGUAGEM MANIPULATIVA

Vamos basear a maior parte da nossa discussão sobre a linguagem manipulativa em um único exemplo, o de matrículas. Suponhamos que a estrutura GIVEN – cuja declaração é

```
DCL 1 GIVEN,
 2 COURSE# CHAR(3),
 2 OFF#    CHAR(3),
 2 EMP#    CHAR(6);
```

contém um conjunto de valores de entrada, representando os dados de matrícula de um empregado em uma determinada oferta de um determinado curso. Queremos verificar se o empregado possui todos os cursos pré-requisito – se sim, aceitaremos a matrícula; caso contrário, vamos rejeitá-la. Apresentaremos procedimentos relacional, hierárquico e de rede para este problema, usando primeiramente operações a nível de registro, e depois operações a nível de conjunto. (Na prática, nossos procedimentos deveriam verificar a validade do número de curso dado, do número da oferta e do número do empregado, observando os registros correspondentes no banco de dados, mas por brevidade omitimos esta etapa.)

A Fig. 27.5 mostra uma solução relacional de um registro por vez, operando sobre a estrutura relacional da Fig. 27.2 (só são necessárias para este problema as relações PREREQ e STUDENT). A lógica do procedimento é a seguinte. Começamos colocando o indicador APPLICATION_OK como *verdadeiro* ('1'B); depois passamos pelos PREREQs do curso dado até que o indicador se torne *falso* ou até que todos esses PREREQs tenham sido processados. Para cada PREREQ examinado, verificamos se há um registro de STUDENT indicando que o empregado dado completou o curso correspondente de pré-requisito; se a verificação falhar, ajustamos APPLICATION_OK para *falso*.

```

APPLICATION_OK = '1'B;
DO PREREQ WHERE PREREQ.COURSE# = GIVEN.COURSE#
    WHILE (APPLICATION_OK);
    IF EXISTS (STUDENT WHERE STUDENT.EMP# = GIVEN.EMP#
        & STUDENT.COURSE# = PREREQ.PREQ#)
    THEN ; /* employee has attended this prereq */
    ELSE APPLICATION_OK = '0'B;
END;
IF APPLICATION_OK THEN
    ALLOCATE STUDENT INITIAL (GIVEN, ' ');

```

Fig. 27.5 Procedimento relacional de um registro por vez.

Se a verificação for bem-sucedida para todos os PREREQs, matricularemos o empregado na oferta especificada, criando um novo registro STUDENT com os dados COURSE#, EMP#, OFF#, e o grau em branco. (Na versão PL/I da UDL, é usado ALLOCATE para criar novos registros.)

Alguns aspectos da codificação da Fig. 27.5 requerem explicação adicional. Primeiro, a expressão PREREQ WHERE ... refere-se a um *conjunto* de registros PREREQ; o loop DO-END é executado uma vez para cada registro desse conjunto (supondo-se que o empregado envolvido tenha participado de todos os cursos pré-requisito). Na *i*-ésima interação, o cursor P (o cursor associado a PREREQs na declaração do tipo de registro PREREQ) é ajustado para indicar a localização do *i*-ésimo PREREQ no conjunto. (A execução bem-sucedida de uma instrução DO ou FIND⁶ faz com que o cursor indique a localização de um registro. O cursor a ser ajustado é especificado por meio da frase “SET (nome-do-cursor)”; se esta frase for omitida, como na Fig. 27.5, será usado por *default* o cursor nomeado na declaração do tipo de registro envolvido.)

Segundo, podemos nos referir ao *i*-ésimo PREREQ (na *i*-ésima interação) por meio da *referência qualificada de cursor* P → PREREQ. A qualificação do cursor é análoga à qualificação hoje no PL/I do indicador de localização; a expressão P → PREREQ é uma referência ao registro PREREQ individual identificado pelo valor corrente do cursor P. (Seria um erro se P não estivesse indicando a localização de um PREREQ.) De forma semelhante, a expressão P → PREREQ.PREQ# é uma referência ao valor PRE# dentro do PREREQ identificado por P. A porção de qualificação do cursor “nome-do-cursor →” pode ser omitida dessas expressões, caso em que (novamente) será usado como *default* o cursor cujo nome está na declaração do tipo de registro envolvido; novamente fizemos uso desta regra de *default* no exemplo (na expressão PREREQ.PREQ# na quinta linha).

Terceiro, a expressão STUDENT WHERE ... também é uma referência a um conjunto. A função integrada EXISTS retorna o valor *verdadeiro* se o seu conjunto argumento não estiver vazio, e *falso* no caso contrário.

Observe, incidentalmente, que a colocação de um cursor indicando a localização de um registro dá ao programador endereçamento direto àquele registro (“referência direta”).

⁶ FIND é usado para a localização de um registro específico. DO é um caminho mais rápido para uma seqüência de FINDs.

Por exemplo, uma vez que o cursor esteja ajustado para um registro PREREQ em particular, o programador pode se referir diretamente àquele registro, e aos campos dentro dele, por meio de referências de cursor qualificado nas quais P é o qualificador do cursor (explícito ou implícito). Não é necessário trazer uma cópia privativa do registro para alguma área local do programa, usando alguma forma de GET ou READ. Em outras palavras, o banco de dados parece estar na memória principal para o programador. Para argumentos que apóiam esta abordagem, veja [27.1].

Vejamos agora a solução hierárquica (Fig. 27.6; a estrutura hierárquica correspondente está na Fig. 27.3).

```
APPLICATION_OK = '1'B;
FIND UNIQUE(COURSE WHERE COURSE.COURSE# = GIVEN.COURSE#);
DO PREREQ UNDER COURSE
    WHILE (APPLICATION_OK);
        IF EXISTS (STUDENT WHERE STUDENT.EMP# = GIVEN.EMP#
            & UNIQUE(COURSE.COURSE# OVER STUDENT)
            = PREREQ.PRE#)
        THEN ; /* employee has attended this prereq */
        ELSE APPLICATION_OK = '0'B;
    END;
    IF APPLICATION_OK THEN
        DO;
            FIND UNIQUE(OFFERING UNDER COURSE
                WHERE OFFERING.OFF# = GIVEN.OFF#);
            ALLOCATE STUDENT INITIAL (GIVEN.EMP#, ' ')
                CONNECT (UNDER OFFERING);
        END;
```

Fig. 27.6 Procedimento hierárquico de um registro por vez.

A lógica é essencialmente a mesma de antes; entretanto, por razões de eficiência, começamos com um FIND do COURSE dado ajustando o cursor C (*default*) para indicá-lo. Esta etapa não é exigida — poderíamos substituir cada uma das referências subsequentes a este COURSE pela expressão completa “UNIQUE (COURSE WHERE . . .)”, se desejado. (UNIQUE é um operador cuja função é a de selecionar o único registro de um conjunto contendo exatamente um registro.) Depois, interagimos com o conjunto de PREREQs sob este COURSE. Observe que os detalhes do teste de EXISTS tornaram-se mais complexos: precisamos comparar o COURSE# do STUDENT sob consideração com o valor de PRE-REQ.PRE#; entretanto, aquele COURSE# não está mais no registro STUDENT, como estava no caso relacional, mas sim no registro único de COURSE sobre aquele STUDENT. Observe também que, uma vez que o registro a ser criado (ALLOCATE) não é uma raiz, temos de alguma forma que identificar o pai ao qual o novo registro deve ser conectado; no exemplo, isto é feito por meio de um FIND àquele pai (um OFFERING), ajustando o cursor O (*default*) para indicá-lo, e depois especificando aquela OFFERING (via a referência qualificada de cursor “O → OFFERING”, com o “O →” assumido por *default*) na opção CONNECT do ALLOCATE.

Podemos ver que, comparado com o procedimento relacional, o procedimento hierárquico requer as seguintes construções adicionais: (a) UNDER, (b) OVER, e (c) CONNECT.

A Fig. 27.7 mostra o procedimento de rede correspondente. As construções adicionais ali, além das da Fig. 27.6, são (a)VIA, e (b)múltiplos UNDERs na opção CONNECT.

Daremos também a solução DBTG ao problema (Fig. 27.8). Neste procedimento assumimos que o SET SELECTION para ATTENDS e HASSTU é BY APPLICATION. É interessante comparar-se as Figs. 27.7 e 27.8, pois ambas representam procedimentos em rede de um registro por vez para o mesmo problema. A Fig. 27.7 usa PL/I enquanto a Fig. 27.8 usa COBOL, mas este fato sozinho não explica as diferenças entre elas.

```
APPLICATION_OK = '1'B;
FIND UNIQUE(COURSE WHERE COURSE.COURSE# = GIVEN.COURSE#);
DO PREREQ UNDER COURSE VIA HASPRE
    WHILE (APPLICATION_OK);
        IF EXISTS (STUDENT WHERE
            UNIQUE(EMPLOYEE.EMP# OVER STUDENT)
                = GIVEN.EMP#
            & UNIQUE(COURSE.COURSE# OVER STUDENT)
                =
            UNIQUE(COURSE.COURSE# OVER PREREQ
                VIA PREOF))
        THEN ; /* employee has attended this prereq */
        ELSE APPLICATION_OK = '0'B;
    END;
    IF APPLICATION_OK THEN
        DO;
            FIND UNIQUE(OFFERING UNDER COURSE
                WHERE OFFERING.OFF# = GIVEN.OFF#);
            FIND UNIQUE(EMPLOYEE
                WHERE EMPLOYEE.EMP# = GIVEN.EMP#);
            ALLOCATE STUDENT INITIAL (' ')
                CONNECT (UNDER OFFERING,
                    UNDER EMPLOYEE);
        END;
    END;
```

Fig. 27.7 Procedimento em rede de um registro por vez.

A Fig. 27.9 resume essas diferenças, e também as diferenças entre as duas soluções em rede e as soluções relacional e hierárquica mostradas anteriormente. (Os números estão baseados nas versões COBOL dos exemplos [mostradas como referência na seção de respostas ao final do livro], embora os números da UDL sejam praticamente independentes de se usar COBOL ou PL/I como linguagem principal. Observe, entretanto, que a definição de "token" é algo arbitrária, e por isso as contagens de *token* devem ser entendidas apenas como medidas grosseiras de tamanhos relativos de procedimentos, e não como números absolutos de mérito. A contagem para DBTG não inclui procedimentos de USE para outras exceções, que são exigidos mas não estão mostrados na Fig. 27.8.)

Incidentalmente, uma das razões primárias para a complexidade da solução DBTG é que o programador tem que manusear o teste de existência central por meio de codificação procedural. Há pelo menos duas estratégias para a execução desse teste: (a) Verificar todos os registros de estudante do empregado envolvido para ver se algum deles é do pré-

```

DECLARATIVES.
USE FOR DB-EXCEPTION ON '0502100'.
EOF-PROC.
MOVE 'YES' TO EOF.
.
.
END DECLARATIVES.

MOVE 'YES' TO APPLICATION-OK
MOVE COURSENO OF GIVEN TO COURSENO IN COURSE
FIND ANY COURSE USING COURSENO IN COURSE
MOVE EMPNO OF GIVEN TO EMPNO IN EMPLOYEE
FIND ANY EMPLOYEE USING EMPNO IN EMPLOYEE
MOVE 'NO' TO EOF
PERFORM UNTIL EOF = 'YES'
    OR APPLICATION-OK = 'NO'
    FIND NEXT PREREQ WITHIN HASPRE
    IF EOF = 'NO'
        FIND OWNER WITHIN PREOF
            RETAINING HASPRE, RECORD CURRENCY
        GET COURSENO IN COURSE
        MOVE COURSENO IN COURSE TO TEMP
        MOVE 'NO' TO APPLICATION-OK
        FIND CURRENT EMPLOYEE
        PERFORM UNTIL EOF = 'YES'
            FIND NEXT STUDENT WITHIN ATTENDS
            IF EOF = 'NO'
                FIND OWNER WITHIN HASSTU
                FIND OWNER WITHIN HASOFFER
                    RETAINING HASPRE, RECORD CURRENCY
                GET COURSENO IN COURSE
                IF COURSENO IN COURSE = TEMP
                    MOVE 'YES' TO APPLICATION-OK
                    MOVE 'YES' TO EOF
                END-IF
            END-IF
        END-PERFORM
        MOVE 'NO' TO EOF
    END-IF
END-PERFORM
IF APPLICATION-OK = 'YES'
    FIND CURRENT COURSE
    MOVE OFFNO OF GIVEN TO OFFNO IN OFFERING
    FIND OFFERING WITHIN HASOFFER CURRENT USING OFFNO IN OFFERING
    FIND CURRENT EMPLOYEE
    MOVE SPACES TO GRADE IN STUDENT
    STORE STUDENT
END-IF

```

Fig. 27.8 Procedimento DBTG.

requisito do curso corrente; (b) Verificar todos os registros de estudantes relativos ao pré-requisito do curso corrente para ver se algum deles é do empregado envolvido. Qual a estratégia mais eficiente, dependerá de muitos parâmetros, incluindo particularmente o número médio de estudantes por curso e o número médio de cursos feitos por empregado. A escolha da estratégia é deixada para o sistema em UDL, mas tem que ser feita pelo programador em DBTG.

	Relacional	Hierárquico	Em rede	DBTG
Banco de dados visto pelo usuário	6 conjuntos de base	6 conjuntos de base + 4 conjuntos agregados	6 conjuntos de base + 7 conjuntos agregados (2 precisam nome)	6 tipos de registros 7 "conjuntos" com nome
Número de tokens no procedimento (Nota 1)	48	71	102	189
Construções da linguagem manipulativa (Nota 2)	FIND WHERE FIND UNDER FIND OVER	FIND WHERE FIND UNDER [VIA] FIND OVER [VIA]	FIND WHERE FIND UNDER [VIA] FIND OVER [VIA]	FIND ANY FIND NEXT WITHIN FIND OWNER WITHIN FIND CURRENT FIND WITHIN USING – SET SELECTION GET RETAIN CURRENCY
	ALLOCATE	ALLOCATE [CONNECT]	ALLOCATE [CONNECT [VIA]]	STORE – SET SELECTION

Nota 1. Um token é uma unidade "léxica" no programa-fonte; por exemplo "MOVE 'NO' TO EOF" consiste de quatro tokens. Todos os números baseiam-se em COBOL.
Nota 2. DO (PERFORM EM COBOL) é equivalente a uma seqüência de operadores FIND WHERE.

Fig. 27.9 Algumas comparações.

Chamamos também a atenção do leitor para a complexidade causada pela noção de corrente no procedimento DBTG. Observe particularmente a necessidade das duas colocações de RETAINING ("RETAINING HASPRE, RECORD CURRENCY" significa "não atualize os indicadores do corrente do conjunto agregado HASPRE ou do tipo de registro a ser encontrado"). É instrutivo verificar em detalhes todo o procedimento. Finalmente, observemos que a situação seria consideravelmente pior se STUDENTS tivessem que ser mantidos em ordem de número de curso dentro do conjunto agregado ATTENDS e em ordem de número de empregado dentro do conjunto agregado HASSTU (esta exigência [razoável] pode ser especificada nas declarações e manuseada pelo sistema em UDL, mas tem que ser cuidada de forma procedural em DBTG).

Apresentaremos agora (Figs. 27.10, 27.11, e 27.12) as soluções UDL a "nível de conjunto" para o problema das matrículas. (Colocamos "nível de conjunto" entre apóstrofos porque os limites entre os níveis de conjunto e de registro são algo arbitrários.) Cada uma dessas soluções consiste de uma única instrução IF. A lógica básica para cada caso é a seguinte: se houver um pré-requisito para um determinado curso que o empregado envolvido não cumpriu, rejeite o pedido de matrícula; caso contrário, efetue a matrícula. Observamos novamente que os dispositivos da linguagem relacional são um subconjunto dos da hierárquica, e estes, por sua vez, um subconjunto dos dispositivos da linguagem em rede.

```

IF EXISTS (PREREQ WHERE PREREQ.COURSE# = GIVEN.COURSE#
    & NOT EXISTS (STUDENT WHERE
        STUDENT.EMP# = GIVEN.EMP#
        & STUDENT.COURSE# = PREREQ.PRE#))
THEN ; /* employee does not have some prereq */
ELSE ALLOCATE STUDENT INITIAL (GIVEN, ' ');

```

Fig. 27.10 Procedimento relacional de um conjunto por vez.

```

IF EXISTS (PREREQ UNDER
    UNIQUE (COURSE WHERE COURSE.COURSE# = GIVEN.COURSE#)
    WHERE NOT EXISTS (STUDENT WHERE
        STUDENT.EMP# = GIVEN.EMP#
        & UNIQUE (COURSE.COURSE# OVER STUDENT)
        = PREREQ.PRE#))
THEN ; /* employee does not have some prereq */
ELSE
    ALLOCATE STUDENT INITIAL (GIVEN.EMP#, ' ')
    CONNECT (UNDER UNIQUE (OFFERING UNDER
        UNIQUE (COURSE WHERE
            COURSE.COURSE# = GIVEN.COURSE#)
        WHERE OFFERING.OFF# = GIVEN.OFF#));

```

Fig. 27.11 Procedimento hierárquico de um conjunto por vez.

```

IF EXISTS (PREREQ UNDER
    UNIQUE (COURSE WHERE COURSE.COURSE# = GIVEN.COURSE#)
        VIA HASPRE
    WHERE NOT EXISTS (STUDENT WHERE
        UNIQUE (EMPLOYEE.EMP# OVER STUDENT)
        = GIVEN.EMP#
        & UNIQUE (COURSE.COURSE# OVER STUDENT)
        =
        UNIQUE (COURSE.COURSE# OVER PREREQ
            VIA PREOF))
THEN ; /* employee does not have some prereq */
ELSE
    ALLOCATE STUDENT INITIAL (' ')
    CONNECT (UNDER UNIQUE (OFFERING UNDER
        UNIQUE (COURSE WHERE
            COURSE.COURSE# = GIVEN.COURSE#)
        WHERE OFFERING.OFF# = GIVEN.OFF#),
        UNDER UNIQUE (EMPLOYEE
            WHERE EMPLOYEE.EMP# = GIVEN.EMP#));

```

Fig. 27.12 Procedimento em rede de um conjunto por vez.

Chamamos sua atenção para o fato de que procedimentos a nível de conjunto nem sempre são mais fáceis de ser escritos ou lidos do que os procedimentos a nível de registro! — muito embora o problema das matrículas seja pouco usual sob este aspecto; anteriormente, neste livro, vimos muitos exemplos em que a solução a nível de conjunto era mais simples para ser entendida. De qualquer forma, mesmo no caso das matrículas, as soluções a nível de conjunto não são pelo menos piores do que as versões a nível de registro; geralmente as operações a nível de conjunto oferecem vantagens sobre as operações a nível de registro — em particular, a vantagem de ter o sistema mais liberdade para escolher o método para implementar as operações (por exemplo, na seleção de caminhos de acesso). Podemos ver também que uma implementação em particular, que está sempre disponível, envolve simplesmente a conversão da determinada operação a nível de conjunto em uma seqüência equivalente de operações a nível de registro.

27.5 DISPOSITIVOS ADICIONAIS

Nesta seção descreveremos rapidamente alguns dispositivos da linguagem manipulativa não discutidos na Seção 27.4.

Cursos limitados — Já vimos diversos exemplos de *referência de conjuntos* — isto é, expressões que representam conjuntos de registros. Suponhamos que precisamos nos referir ao conjunto de STUDENTS que tem um GRADE de 'A', que pode ser escrito na forma

```
S->STUDENT WHERE S->STUDENT.GRADE = 'A'
```

— ou, mais intuitivamente, se S for o cursor *default* de STUDENT, e GRADE (sem qualificação) for não-ambíguo:

```
STUDENT WHERE GRADE = 'A'
```

De qualquer forma, o programador poderá pensar na seguinte avaliação desta instrução: o cursor S é usado para percorrer todo o conjunto de base de STUDENTS, um registro por vez, em alguma seqüência, e os registros que não satisfaçam ao predicado (a condição que se segue ao WHERE) são eliminados. Entretanto, o cursor S em si não é realmente usado, ao invés, é usado um cursor fornecido pelo sistema. De fato, o valor corrente do cursor S é irrelevante, e não é modificado; o símbolo S está sendo usado meramente como elemento sintático para interligar as referências a campos no predicado a casos específicos do tipo de registro envolvido. S, aqui, é um exemplo de *cursor limitado*.

Como outro exemplo, a expressão

```
S1->STUDENT WHERE S1->STUDENT.GRADE = S2->STUDENT.GRADE
```

é uma referência ao conjunto de estudantes que têm o mesmo grau que o estudante selecionado pelo cursor S2. O cursor S2 aqui está executando sua função normal de seleção; em contraste, o cursor S1 está atuando como um cursor limitado.

Expressões de conjuntos — Em edição às expressões que designam conjuntos de registros como declarados, a linguagem também oferece expressões que representam projeções e junções desses registros. Seguem-se alguns exemplos (todos baseados na versão relacional do banco de dados).

1. Número de empregado dos estudantes com grau 'A':

```
STUDENT.EMP# WHERE GRADE = 'A'
```

2. Como em (1), mas com eliminação das duplicatas:

```
DISTINCT (STUDENT.EMP# WHERE GRADE = 'A')
```

3. Junção de professores e alunos sobre número de empregado:

```
(TEACHER, STUDENT) WHERE TEACHER.EMP# = STUDENT.EMP#
```

4. Pares de números de empregado de professores e alunos da mesma oferta do mesmo curso (com as duplicatas eliminadas):

```
DISTINCT ((TEACHER.EMP#, STUDENT.EMP#)
           WHERE TEACHER.COURSE# = STUDENT.COURSE#
             & TEACHER.OFF#      = STUDENT.OFF#)
```

MATCHING – **MATCHING** é um abreviador conveniente para uma forma comum de ocorrência de WHERE. A instrução DO na Fig. 27.5 (a solução relacional a nível de registro) poderia ser escrita de forma equivalente como

```
DO PREREQ MATCHING GIVEN;
```

MATCHING é definido como sendo equivalente a uma cláusula WHERE especificando um conjunto de comparações de igualdade ligadas por AND entre todos os pares de campos que têm o mesmo nome nas duas estruturas (ou registros) consideradas. No exemplo, as duas estruturas são PREREQ e GIVEN, e o único par de campos que têm o mesmo nome é o par PREREQ.COURSE# e GIVEN.COURSE#.

Também é possível especificar-se os campos de coincidência explicitamente, por meio da cláusula ON. Por exemplo, a expressão

```
STUDENT MATCHING TEACHER ON (COURSE#, OFF#)
```

é equivalente à expressão.

```
STUDENT WHERE STUDENT.COURSE# = TEACHER.COURSE#
             & STUDENT.OFF#      = TEACHER.OFF#
```

Observe que a comparação "STUDENT.EMP# = TEACHER.EMP#" não aparece na cláusula WHERE implícita, embora tanto STUDENT como TEACHER tenham um campo chamado EMP#. Se for especificado ON, o conjunto de comparações fica restrito aos campos nomeados. A omissão de ON é equivalente à especificação de uma cláusula ON listando todos os nomes de campos que são comuns às duas estruturas (ou registros) envolvidas.

Uma expressão de conjunto pode incluir WHERE e MATCHING.

FOUND/NOTFOUND – A instrução FIND pode, opcionalmente, incluir uma especificação FOUND/NOTFOUND (análoga ao THEN/ELSE de uma instrução IF). Por exemplo:

```
FIND UNIQUE (COURSE MATCHING GIVEN)
  FOUND  found-unit
  NOTFOUND  notfound-unit
```

A “found-unit” e a “notfound-unit” são “unidades executáveis”, exatamente como no IF-THEN-ELSE. A found-unit é executada se, e somente se, for encontrado o COURSE desejado; a notfound-unit é executada se, e somente se, o curso desejado não for encontrado. Se não for especificado FOUND/NOTFOUND e o registro desejado não for encontrado, surge uma condição de exceção NOTFOUND (veja “manuseio de exceções” abaixo).

Assignment – Como já mencionado anteriormente, a *recuperação*, no sentido de trazer uma cópia do registro a uma área local do programa, não é usualmente necessária, devido ao dispositivo de referência direta. Nas raras ocasiões em que isto for necessário, entretanto, um registro, ou um campo dentro de um registro, pode ser “recuperado” usando-se uma instrução ordinária de designação – por exemplo,

```
ASSIGN S→STUDENT TO STUDENT_AREA;
```

(por razões que fogem ao escopo deste capítulo, usamos uma forma de palavra-chave da instrução *assignment*, ao invés da forma mais familiar “alvo = expressão”). De forma semelhante, um registro, ou um campo dentro de um registro, é *atualizado* por meio de uma instrução de designação – por exemplo,

```
ASSIGN STUDENT_AREA.GRADE TO S→STUDENT.GRADE;
```

FREE – Tal como ALLOCATE é usado para criar novos registros (em PL/I), assim FREE é usado para destruí-los – por exemplo,

```
FREE S→STUDENT;
```

O cursor S é avançado para “pré-selecionar” o próximo STUDENT na seqüência, de tal forma que o FIND NEXT que se segue ao FREE irá operar como a intuição nos está sugerindo – isto é, selecionará o STUDENT que se segue à posição do que recebeu o FREE. Maiores detalhes em [27.1].

Operações com conjuntos agregados – As instruções CONNECT, DISCONNECT, e RECONNECT são fornecidas para criar, destruir, e modificar a interligação entre um dado registro filho e um dado registro pai (via um dado conjunto agregado). Por exemplo (assumindo neste caso que STUDENTS estão como MANUAL e OPCIONAL em relação a OFFERINGS),

```
CONNECT S→STUDENT UNDER O→OFFERING;
DISCONNECT S→STUDENT FROM OFFERING;
RECONNECT S→STUDENT UNDER O→OFFERING;
```

Observe que UNDER especifica a *ocorrência* relevante de registro pai (CONNECT, CONNECT), enquanto que FROM especifica o *tipo* correspondente de pai (DISCONNECT). Cada uma dessas instruções pode incluir adicionalmente uma opção VIA para identificar o conjunto agregado relevante, se necessário ou desejado.

Manuseio de transações — Foi fornecida uma instrução COMMIT para estabelecer pontos de sincronismo e garantir as atualizações ao banco de dados. Uma instrução ROLLBACK foi fornecida para desfazer atualizações não garantidas. O término do programa causa um COMMIT implícito ou um ROLLBACK implícito (dependendo de ter sido o término normal ou anormal).

Manuseio de exceções — Estão definidas diversas condições de exceção na linguagem. Na versão PL/I, elas são manuseadas por meio de várias condições ON—: NOTFOUND, NO-UNIQUE, ADDREX (exceção de endereçamento, que ocorre se, por exemplo, o cursor em uma referência qualificada de cursor não estiver indicando a localização de um registro), e DBERROR ("um coletor"). São também fornecidas funções associadas para depuração, como ONRECTYPE. (ONRECTYPE, por exemplo, tem como seu valor o nome do tipo de registro no qual ocorreu a mais recente exceção.) A ação do sistema para DBERROR é acionar ERROR. A ação do sistema para as outras condições é acionar DBERROR. O programador fica portanto apto a captar exceções em vários níveis diferentes.

Funções integradas — Além de EXISTS, das funções de depuração (ONRECTYPE, etc.), e das "referências integradas" tais como UNIQUE, a UDL oferece funções para contar o número de registros em um conjunto (SETCOUNT), para encontrar o maior valor em um conjunto (SETMAX), para eliminar duplicatas de um conjunto (DISTINCT) e assim por diante.

27.6 CONCLUSÃO

O leitor percebe que fizemos pouco mais do que tocar a superfície da UDL nesta breve descrição. No entanto, foram dados exemplos suficientes para ilustrar a estrutura em camadas e para demonstrar que as porções das linguagens declarativa e manipulativa tornam-se necessariamente mais complexas à medida que a estrutura do banco de dados se torna mais complexa. Fizemos também uma comparação entre a linguagem proposta e a DBTG (que é naturalmente um sistema em rede de um registro por vez). Concluiremos com a seguinte observação: embora a abordagem relacional pareça ser a melhor candidata como base para uma linguagem de banco de dados de uso geral a longo prazo, não há dúvida de que as redes e hierarquias continuarão ainda por aí por algum tempo, pela boa razão de já existir um grande investimento nesses sistemas. Em outras palavras, as três abordagens serão usadas no nível *externo* durante algum tempo. (Enfatizamos que neste capítulo estamos considerando somente o nível externo.) Dado este fato, a idéia de se usar uma linguagem única, bem-estruturada, como interface comum à programação de uma série de sistemas distintos, parece ser muito atrativa; poderia simplificar grandemente os problemas de comunicação entre usuários de diferentes sistemas, facilitar os problemas de educação, e ajudar na migração de programas e programadores de um sistema para outro (incluindo, em particular, a migração de um sistema corrente para um sistema futuro, digamos um relacional). Observe, entretanto, que *não* estamos sugerindo que (digamos) partes relacionais da UDL sejam implementadas como um interface a (digamos) um sistema hierárquico como o IMS. O que estamos sugerindo é que, por exemplo, uma implementa-

ção relacional da UDL (em um sistema relacional) e uma implementação hierárquica (em um sistema hierárquico) possibilitariam um grau útil de semelhança ao nível de linguagem de programação.

EXERCÍCIOS

27.1 Daremos esboços de declaração UDL para uma versão relacional e uma versão em rede do banco de dados de fornecedores-peças-projetos.

```
DCL RSPJ DATABASE /* relational */
BASESETS
(SSET RECTYPE (1 S BASED(CS), 2 S# ...) UNIQUE(S#),
 PSET RECTYPE (1 P BASED(CP), 2 P# ...) UNIQUE(P#),
 JSET RECTYPE (1 J BASED(CJ), 2 J# ...) UNIQUE(J#),
 SPJSET RECTYPE (1 SPJ BASED(CSPJ),
                 2 (S# ..., P# ..., J# ..., QTY ...))
                           UNIQUE ((S#,P#,J#)));
DCL NSPJ DATABASE /* network */
BASESETS
(SSET RECTYPE (1 S BASED(CS), 2 S# ...) UNIQUE(S#),
 PSET RECTYPE (1 P BASED(CP), 2 P# ...) UNIQUE(P#),
 JSET RECTYPE (1 J BASED(CJ), 2 J# ...) UNIQUE(J#),
 SPJSET RECTYPE (1 SPJ BASED(CSPJ), 2 QTY ...)
                           UNIQUE ((S# OVER SPJ,
                                     P# OVER SPJ,
                                     J# OVER SPJ)));
FANSETS
(S_SPJ RECORD(SPJ) UNDER(S)
          UNIQUE((P# OVER SPJ, J# OVER SPJ)),
 P_SPJ RECORD(SPJ) UNDER(P)
          UNIQUE((J# OVER SPJ, S# OVER SPJ)),
 J_SPJ RECORD(SPJ) UNDER(J)
          UNIQUE((S# OVER SPJ, P# OVER SPJ)));
```

Usando essas declarações, dê soluções UDL (relacionais e em redé) aos Exercícios 26.1.1 – 26.1.9. Para as versões em rede, observe cuidadosamente que os conjuntos agregados S_SPJ, P_SPJ, J_SPJ são *essenciais* na estrutura UDL em rede definida acima (o registro SPJ não inclui os campos S#, P#, J#).

27.2 Daremos um esboço das declarações UDL para uma versão relacional e em rede do banco de dados de peças e componentes.

```
DCL RPC DATABASE /* relational */
BASESETS
(PART_SET RECTYPE (1 PART BASED(P), 2 P# ...)
          UNIQUE(P#),
 COMPONENT_SET RECTYPE (1 COMPONENT BASED(C),
                       2 (MAJORP# ...,
                           MINORP# ...,
                           QTY ...))
                           UNIQUE((MAJORP#, MINORP#)));
DCL NPC DATABASE /* network */
BASESETS
(PART_SET RECTYPE (1 PART BASED(P), 2 P# ...)
          UNIQUE(P#),
 COMPONENT_SET RECTYPE (1 COMPONENT BASED(C),
                       2 QTY ...))
```

```
FANSETS
(WU RECORD(COMPARTMENT) UNDER(PART)
    UNIQUE(P# OVER COMPONENT VIA BM),
    BM RECORD(COMPARTMENT) UNDER(PART)
    UNIQUE(P# OVER COMPONENT VIA WU));
```

Usando essas declarações, dê soluções UDL (relacionais e em rede) aos Exercícios 26.4.1 e 26.4.2. No segundo caso, talvez você queira comparar sua solução com a versão SQL (Exercício 8.4).

27.3 Dê uma solução UDL ao problema da Seção 18.6 (dado um número de curso, imprima um relatório contendo todas as ofertas de todos os cursos pré-requisito do curso dado).

27.4 Considere as três soluções de um registro por vez dadas ao problema das matrículas. Para cada uma:

- Projete uma estrutura de memória para representar o banco de dados, mapeando cada registro a um único registro armazenado, mas usando índices, cadeias de indicadores de localização etc., como achar apropriado.
- Derive expressões para representar a quantidade de espaço de memória que cada uma das suas soluções (a) irá requerer. Registre todas as hipóteses em que se basear.
- Derive expressões que representem o número de registros armazenados (registros de dados e de índices) que recebem acesso nos três procedimentos de matrícula. Novamente, registre as hipóteses em que se basear.

REFERÉNCIAS E BIBLIOGRAFIA

27.1 C. J. Date. "An Introduction to the Unified Database Language (UDL)." *Proc. 6th International Conference on Very Large Data Bases* (outubro de 1980).

27.2 C. J. Date. "An Architecture for High-Level Language Database Extensions." *Proc. 1976 ACM SIGMOD International Conference on Management of Data* (junho de 1976).

Uma versão anterior de [27.1].

27.3 C. J. Date. "An Architecture for High-Level Language Database Extension: PL/I Version. Part I: Record-at-a-time Operations." *Proc. SEAS Anniversary Meeting* (setembro de 1977).

27.4 C. J. Date. "An Architecture for High-Level Language Database Extensions (Unified Database Language – UDL): PL/I Version" IBM Technical Report TR 03.099 (junho de 1980).

Uma grande revisão de [27.3].

27.5 C. J. Date. "An Architecture for High-Level Language Database Extensions (Unified Database Language – UDL): COBOL Version." (dezembro de 1978).

27.6 C. J. Date. Relational subset of [27.4] (abril de 1979).

27.7 C. J. Date. Relational subset of [27.5] (abril de 1979).

27.8 SHARE DBMS Language Task Force. "An Evaluation of Three COBOL Data Base Languages – UDL, SQL, and CODASYL." *Proc. SHARE 53* (agosto de 1979).

Na opinião da task force, qualquer linguagem COBOL para bancos de dados deve:

- ser uma extensão natural do COBOL;
- ser fácil para aprender;
- estar de acordo com um padrão;
- suportar relações, hierarquias, e redes;
- promover uma programação de boa qualidade;
- ser utilizável como linguagem de consulta;
- propiciar acesso a nível de conjunto;
- ter uma definição estável;

- aumentar a produtividade dos programadores;
- ter independência de dados;
- refletir posições dos usuários em seu projeto.

27.9 M. H. H. Huits. "Requirements for Languages in Data Base Systems." In [24.3].

Inclui alguns bons exemplos de problemas para os quais as soluções de um registro por vez são mais apropriadas do que as soluções de um conjunto por vez.

Uma Comparação entre as Abordagens Relacional e de Rede

28.1 INTRODUÇÃO

Neste capítulo estaremos considerando os méritos relativos das abordagens relacional e de rede como base para o nível conceitual do sistema. (Não lidaremos explicitamente com hierarquias, tratando-as como meras formas restritas da rede para efeito desta discussão.) Vamos nos concentrar sobre as relações e redes, pois elas estão em competição direta como candidatas a este papel. Não queremos dizer, entretanto, que sejam as únicas candidatas. Sem dúvida, há um consenso generalizado de que nenhuma abordagem é por si adequada à tarefa, mas sim que algum formalismo estendido, como o “modelo entidade-relacionamento” de Chen [28.20], é necessário. Não obstante este fato, ainda é útil examinar-se as duas abordagens com alguma profundidade, pois a maioria dos formalismos estendidos está baseada em uma ou outra.

Embora nossa discussão esteja em sua maior parte baseada em termos do nível conceitual, diversos aspectos são também relevantes para o nível externo.

28.2 O NÍVEL CONCEITUAL

O esquema conceitual tem como objetivo servir como uma fundação sólida e duradoura para a operação global da empresa. Consiste de uma descrição abstrata dos vários tipos de entidades que precisam ser de alguma forma processadas por aquela empresa. Por “sólido e duradouro” queremos dizer que o esquema tem que ser *estável*. Certamente não pode ser dependente das especiarias de qualquer DBMS individual (pode até ter que sobreviver à substituição de um DBMS básico por outro). Mais especificamente, uma determinada entrada, digamos, a descrição de um determinado tipo de entidade, nunca deve ter que mudar uma vez incorporada ao esquema conceitual, *a menos que* ocorra uma mudança na porção do mundo real descrito por aquela entidade específica. Se o esquema conceitual não for estável nesse sentido, as aplicações e o esquema externo também não o serão, levando o usuário à confusão, ao aumento da necessidade de reprogramação, e a maiores chances de erro.

Repetindo: o esquema conceitual não deve sofrer modificações, a menos que algum ajuste no mundo real exija que alguma definição seja também ajustada, para que continue

a refletir a realidade. Naturalmente, um tipo particular de ajustamento freqüentemente necessário é a *expansão* do esquema conceitual para refletir uma porção maior da realidade; veja a discussão sobre crescimento do esquema conceitual na Seção 9.4. No entanto, essa expansão não conflita com o objetivo básico de estabilidade. Como um exemplo de modificação no mundo real que iria requerer uma alteração no esquema conceitual e não apenas uma expansão, consideremos a seguinte mudança na regra que associa empregados e departamentos: sob a regra antiga, cada empregado tinha que pertencer a exatamente um departamento; sob a nova regra, um empregado poderá pertencer simultaneamente a qualquer quantidade de departamentos.

O projeto do esquema conceitual é indubitavelmente a etapa mais importante para a instalação do sistema de bancos de dados. Idealmente, deveria ser a *primeira* etapa (embora, como indicado na Seção 6.4, seja possível preparar o projeto em partes em alguns casos). De qualquer forma, não deve ser influenciado por considerações de como o dado irá ficar fisicamente armazenado e como será o acesso, ou, por outro lado, como será usado em uma aplicação específica. Em outras palavras, o projeto do esquema conceitual deve ser elaborado de forma bastante independente dos esquemas interno e externo associados — pois, se não o for, existe o risco de que o projeto não seja estável e viva sofrendo revisões, com os consequentes impactos sobre os outros componentes do sistema. (Por outro lado, se não forem feitas essas revisões, a instalação pode ficar bloqueada por um esquema conceitual cada vez menos adequado à medida que são colocadas mais e mais aplicações no sistema.)

No entanto, para os sistemas de gerenciamento de bancos de dados existentes hoje, a noção de esquema conceitual independente dos esquemas interno e externo é algo ideal. A maioria dos sistemas correntemente comercializados restringe muito o conjunto de possibilidades disponíveis ao projetista do nível conceitual. Na verdade, como mencionamos no Capítulo 1, a maioria das instalações existentes sequer possui um esquema conceitual; os projetistas atuais simplesmente fornecem um esquema interno e um conjunto de esquemas externos, sendo o “esquema conceitual” nada mais do que a união de todos esses esquemas. (Além disso, a quantidade de possíveis variações entre os esquemas externos e interno é normalmente bastante limitada.) Mas o fato de ser esta a forma tradicional de preparo dos projetos não significa que seja a correta. A experiência tem demonstrado que os problemas mencionados anteriormente (instabilidade, inadequação a novas aplicações) tendem a crescer após algum tempo de uso das instalações [1.12].

Por isso, gostaríamos de frisar que ainda é importante a montagem de um esquema conceitual, com nível de abstração adequado, mesmo que o sistema de gerenciamento disponível seja tal que este esquema exista apenas na forma manuscrita ou datilografada. Se o sistema não suporta um verdadeiro esquema conceitual, o projetista do esquema — presumivelmente o DBA — terá que executar um mapeamento manual do projeto conceitual a uma forma suportada pelo sistema. Se o sistema suportar diretamente o projeto, será naturalmente muito melhor. De qualquer maneira, a empresa estará em situação muito melhor se possuir uma descrição autocontida e sucinta de seus dados operacionais, expressa idealmente em termos que são mais orientados para o homem do que para a máquina, embora precisos. (Cada dia mais se reconhece que o maior obstáculo para o progresso no uso do computador é a dificuldade de *comunicação* entre todo o pessoal envolvido — usuários finais, gerência da empresa, especialistas em programação, administrador do banco de dados etc. [26.1]. O papel que o esquema conceitual desempenha na superação desses problemas é óbvio.)

Expressamos também a esperança de que os sistemas de gerenciamento de bancos

de dados no futuro venham a permitir esquemas conceituais com nível adequado de abstração, suportando assim a técnica de projeto independente advogada acima. Ao mesmo tempo, naturalmente, os sistemas deverão ser capazes de suportar uma grande variedade de esquemas externos, devendo fazê-lo com uma eficiência pelo menos comparável à dos sistemas atuais. No restante deste capítulo estaremos supondo que tal sistema pode e irá, eventualmente, existir.

Como mencionamos no início desta seção, o esquema conceitual não deve ser dependente das peculiaridades de qualquer sistema específico. No entanto, ele tem que ser baseado sobre *alguma* visão dos dados, tal como relações ou redes. Na próxima seção descreveremos algumas propriedades que a visão conceitual dos dados deve ter; depois, nas Seções 28.4 e 28.5, examinaremos relações e redes, uma por vez, para vermos até que ponto elas possuem essas propriedades desejáveis.

28.3 CRITÉRIO PARA O ESQUEMA CONCEITUAL

As duas propriedades mais importantes que a visão conceitual dos dados deve ter são as seguintes.

1. Deve ser a mais simples que a prática permitir.
2. Deve ter uma base teórica palpável.

Vamos verificar cada uma delas.

Simplicidade

Quando dizemos que a visão conceitual tem que ser simples, queremos na verdade significar que ela deve ser fácil de ser entendida e manipulada. Não queremos dizer, de nenhuma forma, que deva ser mínima. (A distinção pode ser melhor esclarecida se fizermos uma analogia com a aritmética. Quando representamos um número na notação posicional familiar, geralmente usamos a decimal como base e não a binária, embora a binária seja logicamente suficiente. A binária é mínima, mas a decimal é mais simples [mais usável] do ponto de vista do usuário.)

A exigência de que a visão seja fácil de ser entendida não requer maiores justificativas. O entendimento é obviamente crucial se se pretende atacar o problema de comunicação mencionado anteriormente. Há naturalmente muitos aspectos relativos à inteligibilidade; relacionamos alguns abaixo.

- A quantidade de construções básicas deve ser pequena.

O esquema conceitual será montado sobre um conjunto de blocos básicos de montagem. É naturalmente desejável que a quantidade de blocos distintos seja mantida numa proporção conveniente e administrável. (Como já mencionado, nós não desejamos no entanto sacrificar a *conclusão* no interesse desse objetivo. A palavra-chave aqui é “conveniente”.)

- Conceitos distintos devem ser claramente separados.

Uma construção individual (bloco de montagem) não deve “englobar” dois ou mais conceitos distintos; pois, se o fizer, torna-se difícil explicar exatamente para que aquela construção está servindo em uma determinada situação (e pode vir a ser usada para uma finalidade para a qual não foi objetivada).

- A simetria deve ser preservada.

Não deve ser necessário representar-se uma estrutura naturalmente simétrica de maneira assimétrica. A simetria é importante para o entendimento. Plagiando Polya (que es-

creveu em um contexto diferente): “Tente tratar simetricamente o que é simétrico, e não destrua arbitrariamente qualquer simetria natural” [28.19].

- A redundância deve ser cuidadosamente controlada.

Provavelmente não deverá ser permitida qualquer redundância, no sentido de aparecer um mesmo fato em dois lugares. (Por “fato” queremos significar a associação entre uma determinada propriedade de uma entidade e aquela entidade — por exemplo, a associação entre um item e seu preço.) No entanto, há outros tipos de redundância [4.1] que não podem ser eliminados. Nesses casos, o esquema conceitual deve incluir uma colocação de qual é exatamente a redundância; veja a discussão sobre redundância controlada no Capítulo 1. (Observemos, de passagem, que correntemente sentimos não ter uma boa definição de redundância; temos somente uma idéia algo vaga de que ela é boa em algumas situações e ruim em outras.)

Também a exigência de que a visão conceitual seja fácil de ser manipulada requer alguma explicação. Embora os usuários não venham a operar diretamente no nível conceitual, eles têm que entender que operações são possíveis naquele nível, pois essas operações são usadas para modelar as transações da empresa. Naturalmente que, em uma implementação em que a visão do usuário seja a mesma ou próxima da visão conceitual, torna-se ainda mais imperativo que essas operações sejam facilmente entendidas. Parafraseando [28.3]: “Alertamos ao leitor para que evite comparar abordagens diferentes somente na base de diferenças nas estruturas de dados que elas suportam. Uma apreciação adequada deve também considerar os tipos de operadores.”

Listaremos alguns aspectos que ajudarão a tornar fácil a manipulação dos dados. Não são necessários comentários adicionais no primeiro caso.

- O número de tipos de operadores deve ser pequeno.
- Devem estar disponíveis operadores de muito alto nível (isto é, potentes).

Nem é preciso dizermos que os operadores têm que ser definidos de forma precisa. No entanto, é também desejável que existam operadores a um nível próximo do impreciso, mas em nível muito alto, como os “operadores” usados na linguagem natural (considere, por exemplo, a transação “Aumente em dez por cento os salários de todos os programadores”). Idealmente, cada transação deve poder ser expressa no nível conceitual de uma, e somente uma forma. Deve ser retirada do usuário a carga de decisões irrelevantes (referente a estratégias de acesso, por exemplo).

- A simetria deve ser preservada.

As transações que possuem uma formulação naturalmente simétrica devem poder ser expressas simetricamente na linguagem manipulativa. Por exemplo, as consultas “Liste todos os empregados que trabalham para o departamento D3” e “Liste todos os empregados que recebem salários de 30.000,00 cruzeiros” devem ter representações similares.

Base teórica

Dada a importância do nível conceitual, é absolutamente essencial que esteja fundamentado em uma sólida base teórica [28.17]. Seu comportamento deve ser totalmente previsível e, até quanto possível, estar de acordo com as expectativas intuitivas dos usuários. Simplesmente não podem ser toleradas surpresas, especialmente as desagradáveis. Qualquer que seja o sistema formal que escolhemos como base para o nível conceitual, temos que ter pleno conhecimento do que é exatamente possível ou não no sistema. Especifica-

mente, temos que estar familiarizados com todas as armadilhas e áreas de problemas, e certos de que não ocorrerão ambigüidade e paradoxo. Em resumo, temos que saber exatamente o que estamos fazendo.

28.4 A ABORDAGEM RELACIONAL

Vejamos agora como a abordagem relacional se encaixa nos requisitos da seção anterior. Primeiro, não há dúvida de que as relações são fáceis de ser entendidas. A quantidade de construções básicas de dados é *uma*, a relação (ou tabela); toda a informação do banco de dados é representada usando-se somente essa construção, sendo esta ao mesmo tempo simples e familiar — as pessoas usam tabelas há séculos. (Lembramos ao usuário que o próprio esquema e todas as outras informações do dicionário podem também ser representadas na forma relacional, como mencionamos no Capítulo 7.) No que toca à separação de conceitos distintos, parece haver pouca, se é que alguma, “mistura” na abordagem relacional.¹ É significativo que a maior parte das pesquisas desde 1970 nas áreas de concorrência, bloqueio, segurança, integridade, definição de visão etc. tenha tomado a abordagem relacional como ponto de partida, precisamente *porque* ela fornece uma base conceitual clara. No que toca a simetria e ausência de redundância, a abordagem relacional novamente parece alcançar os requisitos. [No último caso, a disciplina de normalização garantirá que o mesmo “fato” não apareça em dois lugares.]

As relações são também facilmente manipuladas; podem ser citados os numerosos exemplos deste livro e de outros locais, em apoio a esta afirmativa. Além disso, a afirmativa é verdadeira tanto no nível de uma tupla por vez como no de um conjunto por vez; em outras palavras, estão disponíveis operadores de nível muito alto, bem como os operadores mais familiares de nível baixo. (Os operadores de nível muito alto são os da álgebra relacional e de linguagens equivalentes.) A quantidade de operadores distintos em uma determinada linguagem é pequena, porque há somente um tipo de construção de dados a lidar; essencialmente, precisamos somente de um operador para cada uma das quatro funções básicas de recuperar, inserir, remover, atualizar. Se considerarmos também — como temos que fazer — os operadores necessários para finalidades de autorização e integridade, novamente encontraremos que um conjunto de operadores é tudo que se precisa, pela mesma razão. Por fim, as linguagens relacionais normalmente permitem o que Codd [4.1] chama de “exploração simétrica” — a possibilidade de se ter acesso a uma relação pela especificação de valores conhecidos para qualquer combinação de seus atributos, buscando os valores (desconhecidos) de seus outros atributos. A exploração simétrica é possível porque todas as informações estão representadas em uma mesma forma uniforme.

No tocante à teoria de apoio, a abordagem relacional está não somente baseada firmemente em certos aspectos da teoria matemática de conjuntos, mas também possui um considerável corpo teórico próprio voltado especificamente para aplicação a problemas de bancos de dados. A teoria de normalização discutida no Capítulo 14 fornece um rigoroso conjunto de guias para o projeto do esquema relacional. A teoria de ser o processo relacional completo oferece uma ferramenta valiosa para a medida da potência seletiva de uma

¹ Alguns autores argumentam que as relações *n*-árias misturam diversos *fatos* distintos, e que os objetivos do nível conceitual podem ser melhor atingidos por uma coleção equivalente de relações binárias. Há algum mérito nessa posição. No entanto, consideramos que a distinção entre visões relacionais *n*-árias e binárias é menos significativa do que entre visões relacionais de qualquer tipo e visões em rede tipo DBTG.

linguagem, e para a comparação entre diferentes linguagens candidatas (agora que o conceito está definido, cabe ao projetista da linguagem torná-la completa nesse sentido ou justificar cada afastamento desse objetivo). Sob o tópico de teoria, podemos ainda mencionar o *fechamento* (discutido no Capítulo 12); o resultado de qualquer operação com álgebra relacional ou linguagem equivalente é por si mesmo uma relação, o que nos permite escrever expressões em ninho.² A propriedade de ser fechada é particularmente importante no tocante ao suporte ao usuário não-programador [28.3].

28.5 A ABORDAGEM EM REDE

Antes de qualquer discussão mais detalhada sobre redes, vamos introduzir a importante noção de *essencialidade* [28.3]. A declaração D de uma construção de dados em um esquema S é *essencial* se existir um banco de dados instantâneo B sujeito a S tal que a remoção de B da construção definida por D cause uma perda de informação de B. Ao dizer que ocorreria uma perda de informação, estamos significando precisamente que alguma relação não seria mais derivável.

Apresentaremos alguns exemplos para ilustrar esta idéia, usando uma forma simplificada da sintaxe declarativa da UDL (veja o Capítulo 27).

1. Esquema S1:

```
BASESET COURSE(COURSE#,TITLE)
BASESET OFFERING(COURSE#,OFF#,DATE,LOCATION)
```

As duas declarações são essenciais em S1. (Os dois conjuntos de base são também relações.)

2. Esquema S2:

```
BASESET COURSE(COURSE#,TITLE)
BASESET OFFERING(OFF#,DATE,LOCATION)
FANSET OFFERING UNDER COURSE
```

As três declarações são essenciais em S2. (O primeiro conjunto de base é uma relação; o segundo não é.)

3. Esquema S3:

```
BASESET COURSE(COURSE#,TITLE)
BASESET OFFERING(COURSE#,OFF#,DATE,LOCATION)
FANSET OFFERING UNDER COURSE
      WHERE OFFERING.COURSE# = COURSE.COURSE#
```

As duas declarações de conjuntos de base são essenciais em S3, e a declaração do conjunto agregado não é; não há informação que possa ser derivada deste banco de

²

Incidentalmente, as relações binárias não possuem esta mesma propriedade de fechamento. Por exemplo, a junção de duas relações binárias não é uma relação binária.

dados que não possa ser também derivada dos dois conjuntos de base sozinhos. (Nó-
vamente, os dois conjuntos de base são relações.)

Dada esta noção de essencialidade, podemos agora fazer uma distinção absoluta-
mente crucial entre as abordagens relacional e de rede. Em um esquema relacional, o con-
teúdo total do banco de dados é representado por meio de uma única construção de da-
dos, a relação n -ária.³ Em contraste, em um esquema de rede, há pelo menos um conjunto
agregado carregando essencialidade da informação, pois se assim não fosse o esquema de-
generaria em um esquema relacional com alguns caminhos explícitos.⁴ Em outras pala-
vras, há pelo menos duas construções essenciais de dados na abordagem em rede, o con-
junto de base e o conjunto agregado. No DBTG, em particular, há *cinco* construções de
dados, em que qualquer uma ou todas podem ser usadas para suportar a essencialidade da
informação:

- tipo de registro (corresponde ao conjunto de base);
- conjunto DBTG (corresponde ao conjunto agregado);
- conjunto singular;
- ordenação;
- grupo repetitivo.

Consideraremos agora como a abordagem em rede se encaixa no critério especifica-
do anteriormente para o nível conceitual do sistema.

O primeiro critério foi fácil para ser entendido. Uma comparação entre as Figs. 28.1
(repetição da Fig. 4.4) e 28.2 (versão em rede dos mesmos dados) mostra que, pelo menos
em termos de ocorrências, as redes são algo menos fáceis de ser entendidas do que as rela-
ções. Uma comparação entre as Figs. 27.2 e 27.4 (veja o capítulo anterior) mostra que o
mesmo é verdade para os esquemas. Uma possível razão para o aumento da complexidade
é o aumento da quantidade de construções básicas que o usuário tem que entender para
lidar.

Uma crítica mais forte às redes é de que os conjuntos agregados misturam pelo
menos três conceitos distintos.

1. *Carregam informações* (essencialmente ou não), que são a associação entre os dois
tipos de registros envolvidos.

³ Algumas vezes, permitimos que as relações sejam ordenadas, mas essa ordenação é sempre não-
essencial. Por exemplo, a relação de fornecedores pode ser ordenada em ordem crescente de
número de fornecedor; no entanto, a ordenação é uma mera conveniência – poderíamos ainda
encontrar (digamos) o fornecedor com o terceiro número de fornecedor nessa sequência, mes-
mo que a relação estivesse em uma ordem totalmente randômica.

⁴ O esquema de fornecedores e peças da Fig. 24.16 não contém quaisquer conjuntos de dados
essenciais, não sendo portanto, por esta definição, um esquema de rede. (Entretanto, a não-
essencialidade não ficou explícita nas declarações, ficando o DBMS incapaz de tomar conheci-
mento.) Nossas razões para tornar os conjuntos agregados não-essenciais naquele exemplo foram
dadas no texto; basicamente, o problema é que a manutenção da sequência adequada do filho
é difícil em um conjunto agregado sob DBTG. Mas na verdade isto é uma crítica ao DBTG –
não é um estado de ocorrências que sejam intrínsecas à rede em si.

COMPONENT	MAJOR.P#	MINOR.P#	QUANTITY
	P1	P2	2
	P1	P4	4
	P5	P3	1
	P3	P6	3
	P6	P1	9
	P5	P6	8
	P2	P4	3

Fig. 28.1 Estrutura de componentes e peças: visão relacional.

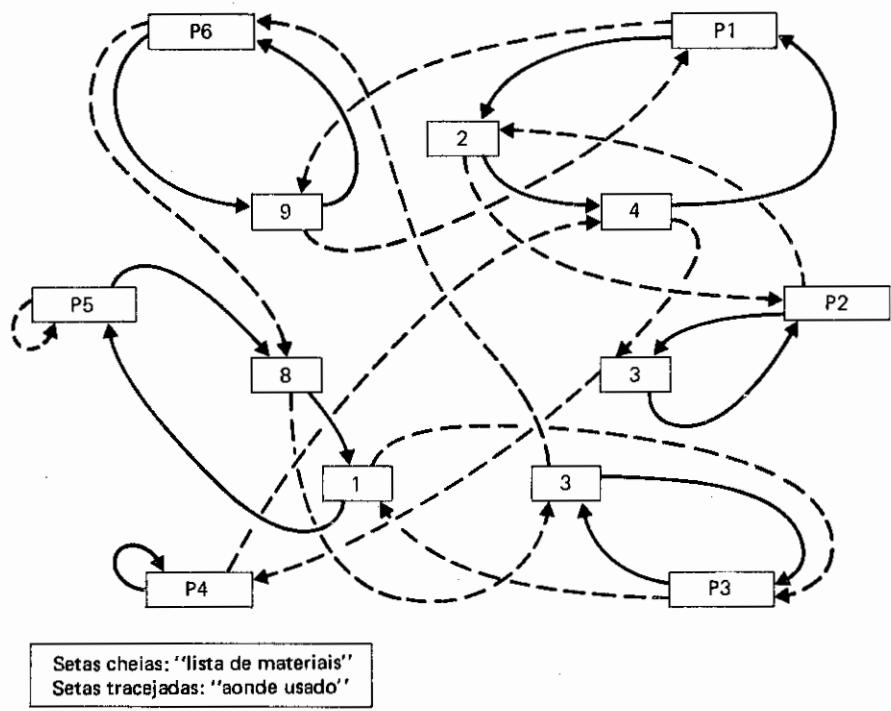


Fig. 28.2 Estrutura de componentes e peças: visão em rede.

2. Fornecem um *caminho de acesso* (na verdade diversos caminhos; pai ao primeiro e ao último filho, filho aos filhos anterior e próximo, filho ao pai).
3. Representam certas *restrições de integridade* (primariamente a restrição de que cada filho possui um pai, embora o conceito de classe de membro permita uma série de variantes sobre esse tema básico).

Adicionalmente, os conjuntos agregados podem ser usados para estabelecer um escoço para fins de autorização; também a ordenação dos filhos dentro de um agregado pode ser usada para carregar informação (novamente essencialmente ou não). Um resultado dessa mistura é que, por exemplo, os programas podem vir a se basear em um caminho de acesso que seja realmente apenas um efeito colateral da forma como o projetista escolheu para representar uma determinada restrição de integridade. Se a restrição de integridade mudar, o esquema terá que ser reestruturado, com um consequente forte impacto sobre a programação correspondente — mesmo que aqueles programas estivessem completamente desinteressados na restrição de integridade propriamente dita. Como um exemplo, convidamos o leitor a considerar o efeito sobre um programa que liste números de empregados por número de departamento (a) usando uma relação ED(EMP#, DEPT#) e (b) usando um conjunto agregado DEPTTEMP (pai DEPT, filho EMP), se a correspondência entre departamento e empregados mudar de um-para-muitos para muitos-para-muitos. (No caso relacional, o pior caso que pode ocorrer é uma mudança trivial no mapeamento externo/conceitual — ED pode ter que ser derivada como uma projeção da relação de empregados. No caso da rede, as mudanças requeridas no nível conceitual são bem mais extensas, e ou o programa ou o mapeamento externo/conceitual exigirá significativo trabalho de reescrita. Se o que for reescrito ficar contido ao mapeamento, incidentalmente, teremos uma situação em que programas estarão usando um caminho de acesso, DEPTTEMP, que não está mais diretamente suportado. Veja o número 2 abaixo.)

Vamos examinar cada um desses três conceitos “misturados” com mais detalhes.

1. Os conjuntos agregados representam certas associações entre entidades. No entanto, em geral, nem todas essas associações serão representadas como conjuntos agregados; provavelmente, nem mesmo todas as associações de um-para-muitos serão assim representadas. Como um exemplo em que não é, consideremos a associação entre cidades e fornecedores. Naturalmente, não há tipo de registro correspondente a cidades; mas se tal tipo de registro fosse adicionado ao esquema seria também adicionado um conjunto agregado com cidades como pais e fornecedores como filhos? Se a resposta for sim, o conjunto agregado será não-essencial, a menos que o tipo de campo cidade seja removido do tipo de registro fornecedor. Esta remoção não é deseável, pelas seguintes razões: (a) os fornecedores têm que ser membros MANUAL do conjunto agregado (para permitir o fato de que o fornecedor exista *antes* que sejam criados os registros de cidade); consequentemente (b) será necessário um novo programa para conectar fornecedores a cidades; e (c) este programa terá que conter o valor da cidade de um determinado fornecedor em *algum lugar* — presumivelmente do campo cidade. Concluímos que a adição de conjuntos agregados essenciais com registros existentes como membros é uma operação não trivial, que levanta questões sobre quão útil é a construção.

2. Os conjuntos agregados representam certos caminhos. No entanto, nem todos esses caminhos são representados por conjuntos agregados; por exemplo, o sistema pode oferecer diversas formas de indexação interna [24.6]. Os programas do usuário não são dependentes da existência desses caminhos de acesso “invisíveis”, mas são definitivamente dependentes da existência dos caminhos visíveis que são representados pelos conjuntos

agregados. (Esta observação mostra que os conjuntos agregados não podem ser olhados apenas como construções puramente lógicas — têm que ser suportados de forma bastante direta no nível físico, pois de outra forma haverá pouca justificativa para se representar exatamente essas associações particulares de uma maneira tão privilegiada. Veja o número 1 acima.) Surge a questão: Por que é esta forma de acesso tornada visível, quando as outras não são?⁵

3. Os conjuntos agregados representam certas restrições de integridade. Entretanto, nem todas essas restrições são representadas por conjuntos agregados; na verdade, a maioria das restrições é especificada em separado da estrutura de dados. Um exemplo dos problemas causados pela mistura dessas restrições já foi dado. Surge a questão: Por que essas restrições particulares recebem este tratamento especial?

Uma pergunta mais geral que surge desses três parágrafos é a seguinte: Como o projetista do esquema decide que associações/caminhos/restrições expressar como conjuntos agregados ou representar de alguma outra forma?

Voltando ao nosso critério de “facilidade de entendimento”, os dois últimos na lista eram *simetria e ausência de redundância*. Um esquema em rede envolvendo conjuntos agregados essenciais tem menos simetria de representação do que um esquema relacional equivalente, pois algumas informações são representadas como registros e algumas como ligações entre registros; por isso tal rede não pode suportar totalmente a “exploração simétrica”. Segundo, um esquema em rede pode certamente ser tão não-redundante quanto um esquema relacional equivalente (mas não mais), mas somente se não envolver quaisquer conjuntos agregados não-essenciais; um conjunto agregado não-essencial contém alguma redundância, pois a associação entre pais e filhos é representada tanto por valores de campos quanto por interligações.

Vejamos agora a *facilidade de manipulação*. Observemos primeiro que cada construção que carrega informação precisa de seu próprio conjunto de operadores para manipulá-la. Portanto, mesmo que nossa atenção fique restrita somente a tipos de registros e conjuntos agregados, vemos que as redes necessariamente requerem mais operadores do que as relações. A linguagem UDL do Capítulo 27 já demonstrou a verdade desta afirmativa. Por exemplo, em DBTG temos STORE para criar um registro e CONNECT para criar uma interligação, ERASE para destruir um registro e DISCONNECT para destruir uma interligação, e assim por diante. (O DBTG não fornece operadores individuais para cada uma das quatro funções básicas de cada construção que carrega informações. Entretanto, isto não significa que tais operadores não sejam necessários — significa simplesmente que nesses casos os usuários têm que programar as funções por si mesmos. Por exemplo, não há um meio direto [operador DML único] para se modificar uma informação que esteja representada por posição dentro de um grupo repetitivo. Considere, por exemplo, o que o usuário tem que fazer para mover o quinto item em um grupo repetitivo para a terceira posição). As observações anteriores são aplicáveis independentemente do nível do operador; assim,

5

Os conjuntos agregados são uma estrutura de aplicação bastante vasta e geral, se consideradas puramente como um mecanismo de acesso e não uma construção lógica. São portanto fortes candidatos para implementação no nível *interno* ou de estrutura de armazenamento (e não no nível conceitual). No entanto, é apropriado que os conjuntos agregados “internos” não usem classes de membros FIXED, MANDATORY, ou AUTOMATIC — tudo deveria ser MANUAL e OPTIONAL. O trabalho de Kay [25.1] apóia essas observações.

podemos certamente fornecer operadores de rede de nível muito alto (um conjunto por vez), mas haverá necessariamente mais desses do que haveria para relações.

Também precisamos de mais operadores de autorização e integridade. Além disso, os controles de autorização e integridade podem se tornar de aplicação bastante complicada. Suponhamos, por exemplo, que dentro do conjunto agregado de departamentos e empregados, os empregados estejam ordenados por valores crescentes do campo salário, e suponhamos que temos um usuário que precisa ver os empregados por departamento — talvez como um caminho de acesso — mas que não tenha acesso à informação sobre salários. Não é suficiente simplesmente que seja omitido o campo salário da visão do usuário; o usuário ainda pode descobrir que Smith ganha mais do que Jones, por exemplo, observando que Smith se segue a Jones na sequência. (Transportadores não-essenciais de informação ainda carregam informações, sendo necessários controles correspondentes.)

Por fim, consideremos a questão da teoria de suporte. Este autor não conhece nenhuma teoria para auxílio ao projetista de um esquema em rede que seja tão completa quanto a teoria da normalização para as relações. É verdade que a teoria da normalização pode ser aplicada aos *registros* da rede, mas somente *após* ter sido decidido que informações serão representadas por registros e quais por outros meios — e esta primeira decisão é naturalmente crítica. As consequências de uma escolha errada causarão instabilidade do esquema. Como um exemplo, considere um esquema em rede representando uma rede de metrô, na qual cada linha esteja representada por um conjunto singular, e a ordem das estações na linha esteja representada pela ordem dos registros no conjunto (um exemplo real de ordem essencial). Suponhamos que em momento posterior seja exigida a incorporação da distância entre estações adjacentes a este esquema. Esta distância é uma propriedade, não de uma estação em si, mas de par de estações adjacentes; entretanto, a adjacência está representada por uma ordenação, não por registros, o que dificulta a introdução do campo de distância. Se introduzirmos um novo “par de estações adjacentes” como tipo de registro, poderemos obviamente usá-lo para conter o campo de distância; no entanto, o esquema existente torna-se então totalmente redundante. (O novo tipo de registro teria que incluir um campo identificando a linha relevante do metrô.) Se colocarmos o campo de distância no tipo existente de registro de estação (mais precisamente, se incorporarmos uma “distância à próxima estação” como campo do tipo de registro de estação, dependendo novamente da ordenação), introduziremos uma assimetria desagradável no esquema. Por exemplo, o algoritmo para computação da distância entre duas estações X e Y de uma dada linha variará significativamente dependendo de se X precede ou se segue a Y na linha. Os problemas são essencialmente devidos ao uso da ordenação como uma construção essencial.

Outra questão teórica importante é a seguinte: É possível suportar um esquema externo relacional sobre um esquema conceitual que envolva conjuntos agregados essenciais? É geralmente aceito que, pelo menos, os usuários não-programadores requerem uma visão relacional do banco de dados. Novamente, este autor não conhece nenhum método completamente geral de suportar tal visão se o esquema conceitual envolver conjuntos agregados essenciais.⁶ Não é difícil criar-se exemplos que mostram que um mapeamento geral

6

Observemos, de passagem, que as propriedades de fechamento das relações não se aplicam aos conjuntos agregados essenciais. Por exemplo, uma “união” de dois agregados de um determinado conjunto agregado que mantenha toda a informação de interligação não é por si outro conjunto agregado.

desse tipo seria bastante complexo. (A informação representada pelo conjunto agregado essencial será representada por uma chave estrangeira no registro filho na visão relacional. Surgem dificuldades se o usuário relacional atualizar esta chave estrangeira ajustando-a para um valor que não case com qualquer registro pai existente. Em algumas situações esta operação tem que ser permitida; em outras não pode ser.)

Algumas questões

Podemos concluir nossa discussão sobre a adequação das redes no nível conceitual invertendo o problema e formulando, mas não respondendo, algumas questões (veja [28.3]). Suponhamos que começamos com relações como sendo a única construção disponível ao nível conceitual. Qual é o efeito de se introduzir novas construções (por exemplo – conjuntos agregados) que carreguem essencialmente informações?

- Do ponto de vista do *sistema* não serão necessários mais operadores? Não se tornam mais complexos os controles de concorrência, autorização e similares? Portanto, torna-se a implementação mais complexa e menos confiável?
- Do ponto de vista do *usuário*, não haverá uma problemática maior na escolha de que operadores usar? Não haverá uma maior variedade possível de erros para se lidar, com um aumento correspondente na variedade de ações remediadoras a serem consideradas?
- Do ponto de vista do *administrador do banco de dados* não haverá muitas escolhas de estruturas possíveis disponíveis? *Há guias confiáveis para se fazer essas escolhas?* (Esta é a questão mais crítica de todas.) Não são os mapeamentos ao nível interno significativamente mais complicados para se definir e manter? Não são as restrições de autorização e integridade mais complicadas para se especificar?

Decorre então que os conjuntos agregados têm que ser *não-essenciais* (por exemplo, para suportar uma visão externa relacional), e então seu papel no nível conceitual tem que ser novamente questionado. Incidentalmente, é interessante observar que o próprio Bachman no seu discurso no Prêmio Turing [26.1] mostra que os conjuntos agregados são fundamentalmente *não-essenciais* e destinados primariamente à melhoria de desempenho: "... O campo de nome 'código de departamento' aparece tanto no registro do empregado como no registro de departamento ... O uso do mesmo valor de dado como chave primária de um registro e chave secundária de um conjunto de registros é o conceito básico sobre o qual estruturas de conjuntos de dados são declaradas e mantidas. . . Com conjuntos de bancos de dados, todos os dados redundantes podem ser eliminados, reduzindo o espaço de memória requerido . . . O desempenho é melhorado . . . onde o dono e alguns ou a maioria dos membros de um conjunto são fisicamente armazenados e recebem acesso em conjunto no mesmo bloco ou página." E mais adiante: "O uso conjunto do código de departamento por ambos [empregado e departamento] os registros e a declaração de um conjunto baseado nesta chave de dados fornecem a base para a criação e manutenção de um relacionamento de conjunto entre um registro de departamento e todos os registros que representam os empregados daquele departamento. [Um benefício dessas construções é] a melhoria significativa de desempenho que advém do uso de conjuntos do banco de dados no lugar de índices primários e secundários para se ter acesso a todos os registros com um valor particular de chave de dados."

28.6 CONCLUSÃO

Nos últimos anos surgiram muitos artigos sobre os méritos relativos das diferentes estruturas de dados e diferentes formas de manipulá-las [28.2–28.17]. Neste capítulo procuramos extraír e apresentar alguns dos temas mais significativos desses artigos. Em particular, buscamos mostrar as vantagens das relações *n*-árias sobre as redes tipo DBTG; entretanto, o leitor encontrará um ponto de vista oposto se pesquisar alguns artigos originais da referência (particularmente [28.8] e [28.12]).

Parece-me apropriado concluir com a colocação de Codd sobre os objetivos da abordagem relacional [9.2]. São:

1. Fornecer alto grau de independência de dados.
2. Fornecer uma visão comunitária dos dados de simplicidade espartana, de forma a que uma grande variedade de usuários de uma empresa (variando do mais simplório em matéria de computação ao mais sofisticado) possa interagir com uma visão *comum* (ao mesmo tempo que não proíbe superposição de visões de usuários para finalidades especiais).
3. Simplificar o trabalho potencialmente enorme do administrador do banco de dados.
4. Introduzir um fundamento teórico (embora modesto) na administração de bancos de dados (um campo que infelizmente está desprovido de princípios sólidos e guias).
5. Unir os campos factuais de administração e recuperação de arquivos, preparando para a adição no futuro de serviços inferenciais no mundo comercial.
6. Elevar a programação de aplicação sobre bancos de dados a um novo nível — um nível em que conjuntos (e mais especificamente relações) são tratados como operandos ao invés de serem processados elemento por elemento.

Ninguém pode proclamar que todos esses objetivos já tenham sido atingidos; resta muito trabalho a ser feito. Entretanto, já foi estabelecida uma forte fundação, e tudo indica haverem boas razões para sermos otimistas com relação ao que eventualmente surja.

REFERÊNCIAS E BIBLIOGRAFIA

- 28.1 R. Rustin (ed.). "Data Models: Data Structure Set versus Relational." *Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control*, Vol. II (maio de 1974).
Anais de um debate ocorrido na conferência SIGMOD de 1974. Inclui referências [28.2–28.5], comentários adicionais de D. C. Tsichritzis e J. R. Lucking, e a transcrição de uma discussão entre um painel e a audiência.
- 28.2 C. W. Bachman. "The Data Structure Set Model." Em [28.1]. Apresenta os argumentos de Bachman de que as duas abordagens são fundamentalmente compatíveis.
- 28.3 E. F. Codd and C. J. Date. "Interactive Support for Non-Programmers: The Relational and Network Approaches." Em [28.1]. A Seção 28.5 do presente capítulo baseou-se fortemente neste artigo.
- 28.4 E. H. Sibley. "On the Equivalence of Data Based Systems". Em [28.1].
- 28.5 C. J. Date and E. F. Codd. "The Relational and Network Approaches: Comparison of the Application Programming Interfaces." Em [28.1].

Um artigo complementar a [28.3]. Algo infeliz ao contrastar uma linguagem relacional de um conjunto por vez (DSL ALPHA) com uma linguagem de rede de um registro por vez (a DML do DBTG).

- 28.6 A. E. Bandurski and D. K. Jefferson. "Data Description for Computer-Aided Design." *Proc. 1975 ACM SIGMOD International Conference on Management of Data* (maio de 1975).

Inclui algumas críticas interessantes tanto às redes como às relações.

- 28.7 A. P. G. Brown. "Modelling a Real World System and Designing a Schema to Represent it." Em [24.3].

Apresenta um conjunto informal de guias para o projeto do esquema DBTG.

- 28.8 C. P. Earnest. "A Comparison of the Network and Relational Data Structure Models." Disponível da Computer Sciences Corporation, 650 N. Sepulveda Blvd., El Segundo, California 90245.

As principais conclusões de Ernest são: "(1) Na prática os dois modelos não são muito diferentes; (2) as estruturas relacionais são um pouco mais simples do que as em rede; mas (3) o preço disto é que o modelo em rede tem mais potência estrutural e *mais*, não menos, independência de dados do que a relacional, sendo por isso mais adequada como base para um padrão."

- 28.9 M. Stonebraker and G. Held. "Networks, Hierarchies, and Relations in Data Base Management Systems." *Proc. 1975 ACM PACIFIC Conference, San Francisco* (abril de 1975). Disponível na ACM Golden Gate Chapter, P. O Box 24055, Oakland, California 94623.

Mostra que o nível da linguagem (um conjunto por vez ou um registro por vez) é mais importante como fator do que a visão básica dos dados.

- 28.10 W. C. McGee. "A Contribution to the Study of Data Equivalence." *Proc. IFIP TC-2 Working Conference on Data Base Management Systems* (eds., Klimbie and Koffeman), abril de 1974. North-Holland (1974).

- 28.11 W. C. McGee. "On the Evaluation of Data Models." *ACM Transactions on Data Base Systems* 1, nº 4 (dezembro de 1976).

Define um conjunto de critérios para a escolha de uma visão particular de dados. Os critérios são: simplicidade, elegância, lógica, apresentação gráfica, modelagem direta, modelagem única, provisão de "esquemas" da estrutura, superposição com modelos co-residentes, possibilidade de particionamento, terminologia consistente, proximidade com a base de implementação, e aplicabilidade de técnicas seguras de implementação. Naturalmente alguns desses critérios chocam-se com outros.

- 28.12 A. Metaxides. "Information-Bearing and Non-Information-Bearing Sets." Em [24.3].

Os termos "carregando informação" e "não carregando informação" no título deste artigo são infelizmente confundidos algumas vezes e usados no lugar de "essencial" e "não-essencial". Como Metaxides corretamente observa, os termos estão mal usados, pois as construções essenciais e não-essenciais carregam ambas informações. O artigo proclama que a eliminação de conjuntos essenciais (a) não oferece benefícios de independência de dados, (b) não oferece benefícios de integridade, (c) realmente não aumenta a simplicidade (Uma simplificação no esquema só é obtida às custas de maior complicaçāo na programação), (d) reduz a flexibilidade, e (e) leva a problemas de projeto e atualizações.

Metaxides era o *chairman* do DBTG na época em que foi produzido o relatório final [23.1].

- 28.13 A. S. Michaels, B. Mittman, and C. R. Carlson. "A Comparison of the Relational and CODASYL Approaches to Data Base Management." *ACM Computing Surveys* 8, nº 1 (março de 1976).

Discute as duas abordagens sob os títulos de definição de dados, manipulação de dados (nível de linguagem, complexidade), proteção de dados, independência de dados, e desempenho. A principal conclusão é que não é desejável nenhuma abordagem única para a administração de bancos de dados (sic) e nenhuma abordagem deverá surgir como dominante em futuro próximo.

- 28.14 G. M. Nijssen. "Data Structuring in DDL and the Relational Data Model." *Proc. IFIP TC-2 Working Conference on Data Base Management Systems* (eds., Klimbie and Koffeman), abril de 1974. North-Holland (1974).

Compara e contrasta as estruturas de dados relacional e em rede, e propõe uma disciplina para usuários de redes. A "DDL" do título é a linguagem de descrição de dados CODASYL. É interessante comparar a disciplina sugerida com outra disciplina destas proposta em [28.3].

28.15 G. M. Nijssen. "Set and CODASYL Set or Coset." Em [24.3].

Considera a DDL CODASYL como uma linguagem para a definição de esquemas conceituais, e sugere várias melhorias à linguagem com este objetivo em mente. As mudanças propostas incluem as seguintes.

- Todos os tipos de registros devem incluir uma chave primária.
- Todos os tipos de conjuntos devem ser não-essenciais.
- Todas as ordenações devem ser não essenciais.
- Um tipo de conjunto deve poder ter o mesmo tipo de registro tanto como dono quanto como membro.
- Um tipo de conjunto não deve poder ter mais do que um tipo de membro.
- O conceito de classe de membro deve ser substituído por uma instrução sobre se a dependência funcional dos donos nos membros é total ou parcial, juntamente com certas restrições de integridade adicionais.

O artigo inclui algumas boas ilustrações sobre por que os conjuntos devem ser não-essenciais. Entretanto, o autor não discute a questão (veja Seção 28.5) "Se os conjuntos têm que ser não-essenciais, para que realmente estarão servindo no esquema conceitual?".

28.16 K. A. Robinson. "An Analysis of the Uses of the CODASYL Set Concept." Em [24.3].

Suporta a contenção da Seção 28.5 de que os conjuntos DBTG não devem aparecer no nível conceitual mas podem ser muito úteis no nível interno.

28.17 T. B. Steel, Jr. "Data Base Standardization: A Status Report." Em [24.3].

Um esboço descritivo da arquitetura ANSI/SPARC, com ênfase no esquema conceitual. O autor argumenta fortemente sobre sua própria convicção de que o único formalismo aceitável para o nível conceitual é o da moderna lógica simbólica.

28.18 E. F. Codd. "Understanding Relations, Instalment nº 4." *SIGMOD bulletin FDT 6*, nº 4 (1974).

Inclui uma descrição muito clara sobre as diferenças entre os seguintes conceitos: (1) o conceito de domínio; (2) comparabilidade de atributos; (3) associação entre uma chave estrangeira e uma chave primária; e (4) o conjunto DBTG ou conjunto agregado. (As diferenças são importantes; é freqüentemente declarado que os conjuntos agregados são o equivalente DBTG de um ou outro dos três primeiros, e não é este o caso.)

28.19 G. Polya. "How To Solve It." Princeton University Press: Princeton Paperback (segunda ed., 1971).

28.20 P. P. S. Chen. "The Entity-Relationship Model – Toward a Unified View of Data." *ACM Transactions on Database Systems* 1, nº 1 (março de 1976).

Respostas a Exercícios Selecionados

CAPÍTULO 1: ARQUITETURA DE SISTEMAS DE BANCOS DE DADOS

- 1.4 A segurança pode ser comprometida
 - (sem bons controles)
- A integridade pode ser comprometida
 - (sem bons controles)
- Pode requerer *hardware* adicional
- A sobrecarga no desempenho pode ser significativa
- É crucial uma operação bem-sucedida
 - (a empresa fica altamente vulnerável a falhas)
- O sistema tende a ser complexo
 - (embora a complexidade deva ser encoberta)

CAPÍTULO 2: ESTRUTURAS DE ARMAZENAMENTO

2.4 Valores gravados em índice Forma expandida

0-2-AB	AB
1-3-CKE	ACKE
3-1-R	ACKR
1-7-DAMS , Tb.	ADAMS , Tb
7-1-R	ADAMS , TR
5-1-O	ADAMSO
1-1-L	AL
1-1-Y	AY
0-7-BAILEY ,	BAILEY ,
6-1-M	BAILEYM

Notas

1. Os dois números que precedem cada valor gravado representam, respectivamente, a quantidade de caracteres frontais que são os mesmos que no valor precedente e a quantidade de caracteres realmente gravados.
2. A forma expandida de cada valor mostra o que pode ser deduzido do índice somente (em uma varredura seqüencial), isto é, sem olhar para o dado.

3. Estamos supondo que o próximo valor do campo indexado não tem BAILEYMS como seus primeiros sete caracteres.

Se tomarmos o *byte* de 8 bits como unidade de espaço de armazenamento e assumirmos que (a) os dois contadores estão acomodados em um único *byte* e (b) cada caractere gravado também requer um único *byte*, a percentagem de economia de espaço de armazenamento é

$$\frac{150 - 35}{150} \cdot 100 = 76.67\%.$$

O algoritmo de pesquisa ao índice é como se segue.

Seja V o valor especificado (completado com brancos se necessário para torná-lo com 15 caracteres de comprimento).

1. Forme a próxima entrada expandida do índice; seja N = comprimento correspondente ($1 \leq N < 15$).
2. Compare a entrada expandida do índice com os N caracteres mais à esquerda de V .
3. Se igual, vá para a etapa 6.
4. Se a entrada do índice for alta, não existe ocorrência de registro armazenado de V ; vá para a saída.
5. Vá para a etapa 1.
6. Recupere a ocorrência de registro armazenado correspondente, e verifique V contra o valor armazenado ali.

Se não houver "próxima" entrada (etapa 1), não existe ocorrência de registro armazenado para V .

Para ACKROYD,S conseguimos um casamento na terceira interação; recuperamos a ocorrência de registro armazenado e encontramos que é a que desejamos.

Para ADAMS,V conseguimos uma "entrada alta no índice" na sexta interação, mostrando que não existe a ocorrência de registro armazenado apropriada.

Para ALLINGHAM,M conseguimos um casamento na sétima interação; entretanto, a ocorrência de registro armazenado recuperada é de ALLEN,S, sendo portanto permissível a inserção do novo para ALLINGHAM,M. (Estamos supondo aqui que o campo indexado é a chave primária, sendo assim únicos os valores.) Isto envolve as seguintes etapas.

1. Encontrar espaço e armazenar a nova ocorrência
2. Ajustar a entrada no índice de ALLEN,S para ter

1-3-LLE

3. Inserir uma entrada no índice entre as de ALLEN,S e AYRES, ST para ter

3-1-I

Observe que a entrada precedente no índice tem que ser alterada. Em geral, a colocação de uma nova entrada no índice afeta a entrada precedente ou a seguinte, ou possivelmente nenhuma – mas nunca ambas.

2.5 O número de *níveis* é o inteiro positivo único k tal que $n^k - 1 < N \leq n^k$. Tomando logaritmos na base n , temos $k - 1 < \log_n N \leq k$; consequentemente

$$k = \text{ceil}(\log_n N),$$

onde $\text{ceil}(x)$ denota o menor inteiro maior do que ou igual a x .

Seja agora B_i o número de *blocos* no i -ésimo nível do índice (onde $i = 1$ corresponde ao nível mais baixo). Mostramos que

$$B_i = \text{ceil}\left(\frac{N}{n^i}\right),$$

e portanto que o número total de blocos é

$$\sum_{i=1}^{k-1} \text{ceil}\left(\frac{N}{n^i}\right).$$

Considere a expressão

$$\text{ceil}\left(\frac{\text{ceil}\left(\frac{N}{n^i}\right)}{n}\right) = x, \text{ digamos}$$

Suponhamos que $N = qn^i + r$ ($0 \leq r \leq n^i - 1$). Então

a) Se $r = 0$,

$$\begin{aligned} x &= \text{ceil}\left(\frac{q}{n}\right) \\ &= \text{ceil}\left(\frac{qn^i}{n^{i+1}}\right) \\ &= \text{ceil}\left(\frac{N}{n^{i+1}}\right). \end{aligned}$$

b) Se $r > 0$,

$$x = \text{ceil}\left(\frac{q+1}{n}\right).$$

Suponhamos que $q = q'n + r'$ ($0 \leq r' \leq n-1$). Então $N = (q'n + r')n^i + r = q'n^{i+1} + (r'n^i + r)$; pois $0 < r \leq n-1$ e $0 \leq r' \leq n-1$,

$$0 < (r'n^i + r) \leq n^{i+1} - (n^i - n + 1) < n^{i+1};$$

$$\text{portanto } \left(\frac{N}{n^{i+1}}\right) = q' + 1.$$

Mas

$$\begin{aligned} x &= \text{ceil}\left(\frac{q'n + r' + 1}{n}\right) \\ &= q' + 1 \end{aligned}$$

pois $1 \leq r' + 1 \leq n$. Portanto em ambos os casos (a) e (b) temos que

$$\text{ceil}\left(\frac{\text{ceil}\left(\frac{N}{n^i}\right)}{n}\right) = \text{ceil}\left(\frac{N}{n^{i+1}}\right).$$

Agora, é imediato que $B_1 = \text{ceil}(N/n)$. É também imediato que $B_{i+1} = \text{ceil}(B_i/n)$, $1 \leq i < k$. Portanto, se $B_i = \text{ceil}(N/n^i)$, então

$$B_{i+1} = \text{ceil}\left(\frac{\text{ceil}\left(\frac{N}{n^i}\right)}{n}\right) = \text{ceil}\left(\frac{N}{n^{i+1}}\right).$$

O restante se segue por indução.

2.6 (a) 3. (b) 6. Por exemplo, se os quatro nomes de campos forem A, B, C, D, e se nós denotarmos um índice pela combinação ordenada apropriada de nomes de campos, serão suficientes os seguintes índices: ABCD, BCDA, CDAB, DABC, ACBD, BDAC, (c) Em geral, o número de índices requeridos é

$${}^N C_n$$

(a quantidade de formas de se selecionar n elementos de um conjunto de N elementos), onde n é o menor inteiro $\geq N/2$. Para uma prova veja Lump [2.13].

CAPÍTULO 3: ESTRUTURAS DE DADOS E OPERADORES CORRESPONDENTES

3.1 Visão relacional:

PESSOA (PERSON)

PNAME	ADDR	---
Arthur	---	---
Bill	---	---
Charlie	---	---
Dave	---	---

HABILIDADE (SKILL)

SNAME	COURSE	JOBCODE	---
Programação	---	---	---
Operação	---	---	---
Engenharia	---	---	---

PERSKIL

PNAME	SNAME	DATE
Arthur	Programação	---
Bill	Operação	---
Bill	Programação	---
Charlie	Engenharia	---
Charlie	Programação	---
Charlie	Operação	---
Dave	Operação	---
Dave	Engenharia	---

3.2 As duas possíveis visões hierárquicas consistem de (a) quatro ocorrências hierárquicas, uma para cada pessoa, com habilidades subordinadas a pessoas; e (b) três ocorrências hierárquicas, uma para cada habilidade, com pessoas subordinadas a habilidades.

3.3 A visão em rede consiste de quatro ocorrências de registros de pessoas, três ocorrências de registros de habilidades, e oito ocorrências de registros "conectores". Cada ocorrência de registro conector representa a conexão entre uma pessoa e uma habilidade, e contém a data em que a pessoa participou do curso correspondente. Cada conector está em uma cadeia de "pessoa" e em uma cadeia de "habilidade".

3.4 (a) Encontre os nomes de todas as pessoas possuindo uma determinada habilidade (digamos S). Isto é muito semelhante às consultas Q1 e Q2 da Fig. 3.2. Para a visão relacional da Resposta 3.1, o procedimento requerido de um registro por vez se parece com os dois procedimentos relacionais da Fig. 3.2; um procedimento algébrico é:

```
SELECT PERSKIL WHERE SNAME=S GIVING TEMP
PROJECT TEMP OVER PNAME GIVING RESULT
```

Para a hierarquia (a) da Resposta 3.2 – pessoas superiores a habilidades – o procedimento requerido segue o mostrado para Q2 na Fig. 3.4; para a hierarquia (b) – habilidades superiores a pessoas – ele

segue o mostrado para Q1 na Fig. 3.4. Para a rede da Resposta 3.3 o procedimento requerido segue o mostrado para Q1 (ou Q2) na Fig. 3.6.

(b) Encontre os nomes de todas as pessoas possuindo pelo menos uma habilidade em comum com uma pessoa especificada (digamos P). Isto é muito mais difícil.

Estrutura relacional da Resposta 3.1

Solução de um registro por vez

```
do until no more PERSKIL records;
    get next PERSKIL where PNAME = P;
        add SNAME to working list;
    end;
position to start of PERSKIL table;
do until no more PERSKIL records;
    get next PERSKIL;
        if SNAME exists in working list
            then merge PNAME into result list
                (eliminating duplicates);
end;
print result list;
```

Observe que precisamos de um operador para “ajustar a posição inicial” (usado antes do segundo *loop*). O procedimento acima poderia ser tornado mais eficiente, e a etapa de “eliminação de duplicatas” tornada desnecessária, se pudéssemos nos basear em que PERSKIL estivesse ordenado em seqüência ascendente de (PNAME, SNAME). Veja o Capítulo 4.

Solução algébrica

```
SELECT PERSKIL WHERE PNAME=P GIVING TEMP1
PROJECT TEMP1 OVER SNAME GIVING TEMP2
JOIN TEMP2 AND PERSKIL OVER SNAME GIVING TEMP3
PROJECT TEMP3 OVER PNAME GIVING RESULT
```

Hierarquia (a) da Resposta 3.2

```
get [next] person where PNAME = P;
do until no more skills
    under this person;
    get next skill
        under this person;
        add SNAME to working list;
end;
position to start of database;
do until no more persons;
    get next person;
    do until no more skills
        under this person;
        get next skill
            under this person;
        if SNAME exists in working list
            then
                do;
```

```
    add PNAME to result list;
    leave inner ("skills") loop;
end;
end;
print result list;
```

Hierarquia (b) da Resposta 3.2

```
do until no more skills;
    get next skill;
    get [next] person
        under this skill;
        where PNAME = P;
    if found
        then add SNAME to working list;
end;
do until no more in working list;
    set S = next SNAME in working list;
    position to start of database;
    get [next] skill where SNAME = S;
    do until no more persons
        under this skill;
        get next person
            under this skill;
        merge PNAME into result list
            (eliminating duplicates);
    end;
end;
print result list;
```

Uma estratégia alternativa é

```
do until no more skills;
    get next skill;
    get [next] person
        under this skill;
        where PNAME = P;
    if found
        then
            do;
                position to start of persons
                    under this skill;
                do until no more persons
                    under this skill;
                    get next person
                        under this skill;
                merge PNAME into result list
                    (eliminating duplicates);
            end;
    end;
```

```

end;
print result list;

Rede da Resposta 3.3
get [next] person where PNAME = P;
do until no more connectors
    under this person;
    get next connector
        under this person;
    get skill
        over this connector;
    add SNAME to working list;
end;
do until no more in working list;
    set S = next SNAME in working list;
    position to start of database;
    get [next] skill where SNAME = S;
    do until no more connectors
        under this skill;
        get next connector
            under this skill;
    get person
        over this connector;
    merge PNAME into result list
        (eliminating duplicates);
end;
end;
print result list;

```

CAPÍTULO 4: ESTRUTURA RELACIONAL DE DADOS

```

4.2 DOMAIN    PNAME      CHARACTER (15) PRIMARY
                ADDR       CHARACTER (40)
                SNAME      CHARACTER (15) PRIMARY
                COURSE     CHARACTER (30)
                JOBCODE    CHARACTER (2)
                DATE       CHARACTER (6)

RELATION    PERSON      (PNAME,ADDR,...)
                PRIMARY KEY (PNAME)
                SKILL       (SNAME,COURSE,JOBCODE,...)
                PRIMARY KEY (SNAME)
                PERSKIL     (PNAME,SNAME,DATE)
                PRIMARY KEY (PNAME,SNAME)

```

Por *default*, está assumido que cada atributo encontra-se definido no domínio que tem o mesmo nome.

CAPÍTULO 6: ESTRUTURA DE DADOS DO SISTEMA R

```

6.1 CREATE TABLE PERSON  ( PNAME   ( CHAR(15), NONULL ),
                           ADDR    ( CHAR(40) ), ... )

```

```

CREATE TABLE SKILL   ( SNAME   ( CHAR(15), NONULL ),
                      COURSE  ( CHAR(30) ),
                      JOBCODE ( CHAR(2) ), ... )
CREATE TABLE PERSKIL ( PNAME   ( CHAR(15), NONULL ),
                      SNAME   ( CHAR(15), NONULL ),
                      DATE    ( CHAR(6) )      )
CREATE UNIQUE INDEX XP ON PERSON  ( PNAME )
CREATE UNIQUE INDEX XS ON SKILL   ( SNAME )
CREATE UNIQUE INDEX XPS ON PERSKIL ( PNAME, SNAME )

```

CAPÍTULO 7: MANIPULAÇÃO DE DADOS DO SISTEMA R

A maior parte das respostas a seguir não é única

- 7.1 SELECT *
FROM J
- 7.2 SELECT *
FROM J
WHERE CITY='LONDON'
- 7.3 SELECT P#
FROM P
WHERE WEIGHT=
(SELECT MIN(WEIGHT)
FROM P)
- 7.4 SELECT UNIQUE S#
FROM SPJ
WHERE J#='J1'
- 7.5 SELECT S#
FROM SPJ
WHERE P#='P1' AND J#='J1'
- 7.6 SELECT JNAME
FROM J
WHERE J# IN
(SELECT J#
FROM SPJ
WHERE S#= 'S1')
- 7.7 SELECT UNIQUE COLOR
FROM P
WHERE P# IN
(SELECT P#
FROM SPJ
WHERE S#= 'S1')
- 7.8 SELECT S#
FROM SPJ
WHERE J#='J1'
AND S# IN
(SELECT S#
FROM SPJ
WHERE J#= 'J2')
- 7.9 SELECT UNIQUE S#
FROM SPJ
WHERE J#= 'J1'

AND P# IN
(SELECT P#
FROM P
WHERE COLOR='RED')

7.10 SELECT UNIQUE P#
FROM SPJ
WHERE J# IN
(SELECT J#
FROM J
WHERE CITY='LONDON')

7.11 SELECT UNIQUE S#
FROM SPJ
WHERE J# IN
(SELECT J#
FROM J
WHERE CITY='LONDON'
OR CITY='PARIS')
AND P# IN
(SELECT P#
FROM P
WHERE COLOR='RED')

7.12 SELECT UNIQUE P#
FROM S,SPJ,J
WHERE S.S#=SPJ.S#
AND SPJ.J#=J.J#
AND S.CITY=J.CIT

7.13 SELECT UNIQUE P#
FROM SPJ
WHERE J# IN
(SELECT J#
FROM
WHERE CITY='LONDON')
AND S# IN
(SELECT S#
FROM S
WHERE CITY='LONDON')

7.14 SELECT J.J#
FROM S,SPJ,J
WHERE S.S#=SPJ.S#
AND SPJ.J#=J.J#
AND S.CITY=J.CITY

7.15 SELECT J#
FROM J
WHERE NOT EXISTS
(SELECT *
FROM S,SPJ,P
WHERE S.S#=SPJ.S#
AND P.P#=SPJ.P#
AND J.J#=SPJ.J#
AND S.CITY='LONDON'
AND P.COLOR='RED')

7.16 SELECT UNIQUE S#

 FROM SPJ

 WHERE P# IN

 (SELECT P#

 FROM SPJ

 WHERE S# IN

 (SELECT S#

 FROM SPJ

 WHERE P# IN

 (SELECT P#

 FROM P

 WHERE COLOR='RED')))

7.17 SELECT UNIQUE J#

 FROM SPJ

 WHERE P# IN

 (SELECT P#

 FROM SPJ

 WHERE S#='S1')

7.18 SELECT UNIQUE S.CITY,J.CITY

 FROM S,SPJ,J

 WHERE S.S#=SPJ.S#

 AND J.J#=SPJ.J#

7.19 SELECT S.CITY,P#,J.CITY

 FROM S,SPJ,J

 WHERE S.S#=SPJ.S#

 AND J.J#=SPJ.J#

7.20 SELECT S.CITY,P#,J.CITY

 FROM S,SPJ,J

 WHERE S.S#=SPJ.S#

 AND J.J#=SPJ.J#

 AND S.CITY=J.CITY

7.21 SELECT UNIQUE S#

 FROM SPJ SPJX

 WHERE EXISTS

 (SELECT *

 FROM SPJ SPJY

 WHERE NOT EXISTS

 (SELECT *

 FROM SPJ SPJZ

 WHERE NOT EXISTS

 (SELECT *

 FROM SPJ

 WHERE S#=SPJX.S#

 AND P#=SPJY.P#

 AND J#=SPJZ.J#)))

7.22 SELECT UNIQUE J#

 FROM SPJ SPJX

 WHERE NOT EXISTS

 (SELECT *

 FROM SPJ

 WHERE J#=SPJX.J#

 AND S#=S1)

7.23 SELECT UNIQUE P#
FROM SPJ SPJX
WHERE NOT EXISTS
(SELECT *
FROM J
WHERE CITY='LONDON'
AND NOT EXISTS
(SELECT *
FROM SPJ
WHERE J#=J.J#
AND P#=SPJX.P#))

7.24 SELECT UNIQUE J#
FROM SPJ SPJX
WHERE NOT EXISTS
(SELECT *
FROM SPJ SPJY
WHERE S#='S1'
AND NOT EXISTS
(SELECT *
FROM SPJ
WHERE P#=SPJY.P#
AND J#=SPJX.J#))

7.25 SELECT UNIQUE J#
FROM SPJ SPJX
WHERE NOT EXISTS
(SELECT *
FROM SPJ SPJY
WHERE J#=SPJX.J#
AND NOT EXISTS
(SELECT *
FROM SPJ
WHERE P#=SPJY.P#
AND S#='S1'))

7.26 SELECT UNIQUE J#
FROM SPJ SPJX
WHERE NOT EXISTS
(SELECT *
FROM SPJ SPJY
WHERE EXISTS
(SELECT *
FROM SPJ
WHERE S#='S1'
AND P#=SPJY.P#)
AND NOT EXISTS
(SELECT *
FROM SPJ
WHERE S#='S1'
AND P#=SPJY.P#
AND J#=SPJX.J#))

7.27 SELECT UNIQUE J#
FROM SPJ SPJX
WHERE NOT EXISTS
(SELECT *
FROM SPJ SPJY

```

        WHERE EXISTS
          (SELECT *
           FROM SPJ
           WHERE P#=SPJY.P#
           AND J#=SPJX.J#)
      AND NOT EXISTS
        (SELECT *
         FROM SPJ
         WHERE S#= 'S1'
         AND P#=SPJY.P#
         AND J#=SPJX.J#))

7.28 SELECT UNIQUE J#
      FROM SPJ SPJX
      WHERE NOT EXISTS
        (SELECT *
         FROM SPJ SPJY
         WHERE P# IN
           (SELECT P#
            FROM P
            WHERE COLOR='RED')
        AND NOT EXISTS
          (SELECT *
           FROM SPJ
           WHERE S#=SPJY.S#
           AND J#=SPJX.J#))

7.29 UPDATE J
      SET JNAME='VIDEO'
      WHERE J#= 'J6'

7.30 UPDATE P
      SET COLOR='ORANGE'
      WHERE COLOR='RED'

7.31 DELETE SPJ
      WHERE P# IN
        (SELECT P#
         FROM P
         WHERE COLOR='RED')
    DELETE P
      WHERE COLOR='RED'

7.32 SELECT COUNT(UNIQUE J#)
      FROM SPJ
      WHERE S#= 'S3'

7.33 SELECT SUM(QTY)
      FROM SPJ
      WHERE S#= 'S1'
      AND P#= 'P1'

7.34 SELECT P#,J#,SUM(QTY)
      FROM SPJ
      GROUP BY P#,J#

```

CAPÍTULO 8: SQL EMBUTIDA

8.1 Há basicamente duas maneiras de se escrever tal programa. A primeira envolve dois cursores, digamos CS e CP, definidos como se segue:

```

$LET CS BE SELECT * INTO $$#, ...
    FROM S
    ORDER BY S#;
$LET CP BE SELECT * INTO $P#, ...
    FROM P
    WHERE P# IN
        (SELECT P# FROM SPJ
        WHERE S# = $$#)
    ORDER BY P#;

```

A lógica neste caso é essencialmente como se segue:

```

$OPEN CS;
DO for all suppliers;
$FETCH CS;
print S;
$OPEN CP;
DO for all parts for this supplier;
$FETCH CP;
print P;
END;
END;

```

A segunda abordagem usa um único cursor:

```

$LET C BE SELECT * INTO ...
    FROM S,P,SPJ
    WHERE S.S# = SPJ.S#
    AND SPJ.P# = P.P#
    ORDER BY S.S#,P.P#;

```

Lógica:

```

$OPEN C;
DO for all joined records;
$FETCH C;
IF S.S# different from previous iteration
THEN print S information;
print P information;
END;

```

8.2 Suponhamos que o programa inclui uma instrução LET da forma

```
$LET C BE SELECT ... FROM T ...;
```

O otimizador do RDS é responsável pela escolha de um caminho de acesso correspondente ao cursor C. Suponhamos que escolha um índice baseado no campo F da tabela T. O conjunto de registros aos quais se pode ter acesso via C quando C está ativo será então ordenado de acordo com os valores de F. Se o programador fosse autorizado a fazer um UPDATE de um valor de F via o cursor C – isto é, via uma instrução UPDATE da forma

```
$UPDATE T SET F = ... WHERE CURRENT OF C;
```

– então o registro atualizado teria provavelmente que ser “movido” (logicamente falando), porque agora pertenceria a um local diferente em relação à ordenação do conjunto. Em outras palavras, o cursor C efetivamente saltaria para uma nova posição, com resultados imprevisíveis. Para evitar essa situação, o usuário deve alertar ao otimizador quanto a quaisquer campos a serem atualizados, para que os caminhos de acesso baseados naquele campo *não* sejam escolhidos.

8.3 Daremos aqui um esboço de uma possível solução. (Não proclamamos que esta solução seja muito eficiente. Como pode ser melhorada?)

```
GET LIST (GIVENP#);
print GIVENP#;
$BEGIN TRANSACTION;
CALL RECURSION (GIVENP#);
$END TRANSACTION;
RETURN;

RECURSION:
PROC (UPPER_PART) RECURSIVE;
$DCL UPPER_PART;
$DCL LOWER_PART INITIAL (' ');
$LET C BE SELECT MINORP#
    INTO $LOWER_PART
    FROM COMPONENT
    WHERE MAJORP# = $UPPER_PART
    AND MINORP# > $LOWER_PART
    ORDER BY MINORP#;

DO forever;
$OPEN C;
$FETCH C;
IF not found THEN RETURN;
IF found THEN DO;
    print LOWER_PART;
    $CLOSE C;
    CALL RECURSION (LOWER_PART);
END;
END;
END; /* of RECURSION */
```

Observe que o mesmo cursor, C, é usado a cada invocação de RECURSION. Não há forma em SQL de se criar novas "instâncias" de um cursor dinamicamente. (Em contraste, são criadas dinamicamente novas instâncias de UPPER_PART e LOWER_PART cada vez que RECURSION é invocado; essas instâncias são destruídas ao se completar a invocação.) Devido a este fato, temos que usar um artifício ("... AND MINORP# > \$LOWER_PART ORDER BY MINORP#") para que, a cada invocação de RECURSION, ignoremos todos os componentes imediatos (LOWER_PARTs) do UPPER_PART corrente que foi processado.

Para algumas abordagens alternativas e discussão sobre este problema, veja a referência [5.10].

8.4 No Sistema R, cada variável alvo em uma cláusula INTO pode ter associada uma *variável indicador*, especificada como se segue:

```
INTO alvo [:indicador] [, alvo [:indicador]] ...
```

Caso algum campo particular a ser recuperado seja nulo, o indicador correspondente é ajustado para um valor negativo. Se não for especificado indicador, SYR_CODE é ajustado para um valor negativo.

CAPÍTULO 9: O NÍVEL EXTERNO DO SISTEMA R

9.1 O problema aqui é como seria definido o campo SP.QTY? A resposta sensata parece ser que, para um dado par (S#, P#), SP.QTY deva ser a *soma* de todos os valores SPJ.QTY, tomados sobre todos os J#'s para aquele par (S#, P#).

```
DEFINE VIEW SP (S#, P#, QTY)
AS SELECT S#, P#, SUM(QTY)
FROM SPJ
GROUP BY J#
```

A tabela SP não pode ser atualizada.

9.2 Daremos um exemplo simples para mostrar como DEFINE VIEW pode ser estendida para definir visões hierárquicas. O exemplo define uma versão hierárquica da visão V2 (veja a Seção 9.3), consistindo de um tipo de registro PART como raiz e outro tipo de registro LOC como dependente daquela raiz. PART contém um único campo (P#), bem como LOC (CITY).

```
DEFINE VIEW HV2
  ( PART (P#) OVER (
    LOC (CITY) . . )
  AS SELECT P#, CITY
    FROM SP, S
    WHERE SP.S# = S.S.#
```

9.3 Para operações de recuperação, é válida qualquer hierarquia derivável. Para operações de atualização há muitas restrições. Não daremos uma solução completa ao problema, contentando-nos com um exemplo ilustrativo. Consideremos a seguinte hierarquia (fornecedores sobre peças sobre projetos):

```
DEFINE VIEW HSPJ
  ( HS (S#,CITY)
    OVER ( HP (P#)
      OVER ( HJ (J#,QTY) ) )
  AS . . .
```

que suporemos estar derivada de maneira óbvia do esquema relacional de fornecedores-peças-projetos. Observemos que o tipo de registro raiz (HS) nesta hierarquia é essencialmente idêntico à relação S básica, exceto que foram omitidos certos atributos (não a chave primária). Teria sido também possível omitir-se tuplas individuais (as que não satisfizessem a algum predicado definidor). Observemos também que cada tipo de registro sob a raiz (isto é, HP e HJ) consiste de uma chave primária de alguma outra relação básica juntamente com zero ou mais atributos adicionais que são "totalmente funcionalmente dependentes" (veja o Capítulo 14) na "chave totalmente concatenada" (veja o Capítulo 16) daquele tipo de registro. Dada esta estrutura, podem ser feitas atualizações sobre cada um dos três tipos de registros HS, HP, HJ; qualquer dessas atualizações pode ser mapeada como atualização equivalente das relações básicas, *exceto*: (a) Uma HJ só pode ser removida como efeito colateral da remoção de uma HP ou HS; (b) uma HP só pode ser inserida se for simultaneamente inserida uma HJ subordinada.

Observemos de passagem que este exemplo ilustra alguns guias que podem ser úteis no projeto de um esquema hierárquico. Como no caso do guia do sistema relacional (veja o Capítulo 14), um princípio orientador é "um fato em um lugar" – isto é, evitar redundância. Entretanto, estes guias não são tão completos como os para as relações. Por exemplo, não indicam se a hierarquia deve ser com fornecedores sobre peças sobre projetos, como acima, ou fornecedores sobre projetos sobre peças, ou fornecedores sobre peças e projetos (nessa ordem), ou uma das outras nove possibilidades. Nem indicam se a hierarquia deve conter somente informações "SPJ", ou se a raiz, pelo menos, deve conter atributos adicionais para o tipo de entidade envolvida. E finalmente, qualquer que seja a hierarquia selecionada, terá ainda que ser suplementada por duas *relações* para o tipo de entidade não-raiz.

CAPÍTULO 11: QUERY BY EXAMPLE

No que se segue, numeramos as questões com 11.*n*, onde *n* é o número do exercício original no Capítulo 7.

11.1

J	J#	JNAME	CITY
P.			

11.2

J	J#	JNAME	CITY
P.			LONDON

11.3

P	P#	PNAME	COLOR	WEIGHT	CITY
7	P.PX			W -< W	

11.4

SPJ	S#	P#	J#	QTY
	P.SX		J1	

11.5

SPJ	S#	P#	J#	QTY
	P.SX	P1	J1	

11.6

J	J#	JNAME	CITY	SPJ	S#	P#	J#	QTY
	JX	P.JN			S1		JX	

11.7

P	P#	PNAME	COLOR	WEIGHT	CITY	SPJ	S#	P#	J#	QTY
	PX		P.COL				S1	PX		

11.8

SPJ	S#	P#	J#	QTY
	P.SX		J1	
	SX		J2	

11.9

SPJ	S#	P#	J#	QTY	P	P#	PNAME	COLOR	WEIGHT	CITY
	P.SX	PX	J1			PX		RED		

11.10

SPJ	S#	P#	J#	QTY	J	J#	JNAME	CITY
		P.PX	JX			JX		LONDON

11.11

P	P#	PNAME	COLOR	WEIGHT	CITY
	<u>PX</u>		RED		
	<u>PY</u>		RED		

J	J#	JNAME	CITY
	<u>JX</u>		LONDON
	<u>JY</u>		PARIS

SPJ	S#	P#	J#	QTY
	<u>P.SX</u>	<u>PX</u>	<u>JX</u>	
	<u>P.SY</u>	<u>PY</u>	<u>JY</u>	

observe que são necessários dois exemplos distintos de números de peças.

11.12

S	S#	SNAME	STATUS	CITY
	<u>SX</u>			<u>CX</u>

J	J#	JNAME	CITY
	<u>JX</u>		<u>CX</u>

SPJ	S#	P#	J#	QTY
	<u>SX</u>	<u>P.PX</u>	<u>JX</u>	

11.13

S	S#	SNAME	STATUS	CITY
	<u>SX</u>			LONDON

J	J#	JNAME	CITY
	<u>JX</u>		LONDON

SPJ	S#	P#	J#	QTY
	<u>SX</u>	<u>P.PX</u>	<u>JX</u>	

11.14

S	S#	SNAME	STATUS	CITY
	<u>SX</u>			<u>C</u>

J	J#	JNAME	CITY
	<u>JX</u>		<u>C</u>

SPJ	S#	P#	J#	QTY
	<u>SX</u>		<u>P.JX</u>	

11.15

S	S#	SNAME	STATUS	CITY
	<u>SX</u>			LONDON

P	P#	PNAME	COLOR	WEIGHT	CITY
	<u>PX</u>		RED		

SPJ	S#	P#	J#	QTY
<u> </u>	<u>SX</u>	<u>PX</u>	<u>JX</u>	<u>P.JX</u>

11.16

SPJ	S#	P#	J#	QTY
	<u>P.SX</u>	<u>PX</u>		
	<u>SY</u>	<u>PX</u>		
	<u>SY</u>	<u>PY</u>		

P	P#	PNAME	COLOR	WEIGHT	CITY
	<u>PY</u>		RED		

11.17

SPJ	S#	P#	J#	QTY
	<u>S1</u>	<u>PX</u>		

PX

PX

P.JX

11.18

S	S#	SNAME	STATUS	CITY
	<u>SX</u>			<u>SC</u>

J	J#	JNAME	CITY
	<u>JX</u>		<u>JC</u>

SPJ	S#	P#	J#	QTY
	<u>SX</u>		<u>JX</u>	

RESULT	SCITY	JCITY
	<u>P.SC</u>	<u>P.JC</u>

11.19

S	S#	SNAME	STATUS	CITY
	<u>SX</u>			<u>SC</u>

J	J#	JNAME	CITY
	<u>JX</u>		

SPJ	S#	P#	J#	QTY
	<u>SX</u>	<u>PX</u>	<u>JX</u>	

RESULT	SCITY	P#	JCITY
<u>P.</u>	<u>SC</u>	<u>PX</u>	<u>JC</u>

11.20 Acrescente à solução 11.19:

CONDITIONS	
SC	$\neg =$ JC

11.29

J	J#	JNAME	CITY
	J6	U. VIDEO	

11.30

P	P#	PNAME	COLOR	WEIGHT	CITY
	PX		RED		
U.	PX		ORANGE		

11.31

P	P#	PNAME	COLOR	WEIGHT	CITY
D.			RED		
	PX		RED		

SPJ	S#	P#	J#	QTY
D.		PX		

11.32

SPJ	S#	P#	J#	QTY
	S3		P.CNT.UNQ.ALL.JX	

11.33

SPJ	S#	P#	J#	QTY
	S1	P1		P.SUM.ALL.QX

SPJ	S#	P#	J#	QTY
		P.G.PX	P.G.JX	P.SUM.ALL.QX

Incidentalmente, se quiséssemos o resultado mostrado em ordem dos J# dentro da ordem de P# aqui, poderíamos ter especificado "AO(1)." entre o "P." e o "G." para P#, e "AO(2)." entre o "P." e o "G." para J#.

CAPÍTULO 12: ÁLGEBRA RELACIONAL

$$\begin{aligned} 12.1 \text{ (a) } A \text{ INTERSECT } B &= A \text{ MINUS } (A \text{ MINUS } B) \\ &= B \text{ MINUS } (B \text{ MINUS } A) \end{aligned}$$

$$\text{(b) } A \text{ DIVIDE BY } B$$

(onde A tem atributos X, Y e B tem atributo Z-X, Y, e Z possivelmente composto)

$$= (A \text{ MINUS } ((A[X] \text{ TIMES } B) \text{ MINUS } A)) [X]$$

12.2 Sejam A e B dois membros quaisquer de um dado conjunto de relações com nomes. Podem ser derivadas relações adicionais (sem nome) por meio de expressões algébricas sem ninhos envolvendo exatamente um dos operadores algébricos e um ou ambos, como for apropriado, de A ou B. Para cada uma dessas expressões sem ninhos é bastante direto encontrar-se uma expressão SQL semanticamente equivalente, como mostrado abaixo. (A notação se propõe ser auto-explicativa)

Algebra	SQL
A UNION B	SELECT all-columns-of-A FROM A UNION SELECT all-columns-of-B FROM B
A MINUS B	SELECT all-columns-of-A FROM A WHERE NOT EXISTS (SELECT * FROM B WHERE all-columns-of-A = all-columns-of-B)
A TIMES B	SELECT all-columns-of-A,all-columns-of-B FROM A,B
A WHERE p	SELECT all-columns-of-A FROM A WHERE p
A[x]	SELECT x FROM A (We ignore UNIQUE.)

Uma vez que a SQL nos permite guardar (e dar nome) ao resultado de qualquer consulta, um procedimento em linha com o que se encontra acima é suficiente para demonstrar que a linguagem é completa no sentido original de Codd [12.1]. Entretanto, indicaremos abaixo como se pode mostrar que é completa em um sentido mais abrangente, do que qualquer relação derivável via uma única expressão algébrica é derivável via uma única expressão SQL. O esboço da prova é o seguinte.

Etapa 1. (Já feita). Mostramos que, se A e B são dois membros quaisquer de um dado conjunto de relações com nomes, qualquer relação derivável via uma única expressão algébrica sem ninhos envolvendo exatamente um dos operadores algébricos e A e/ou B pode ser derivada via uma única expressão SQL.

Etapa 2. Sejam agora A e B duas relações quaisquer deriváveis do conjunto dado de relações com nomes via expressões algébricas possivelmente com ninhos. Podem ser derivadas relações adicionais (sem nome) de A e B por meio de expressões envolvendo exatamente um dos operadores algébricos aplica-

do sobre um ou ambos, como for apropriado, de A ou B. Mostraremos que *se* existirem expressões SQL representando A e B, *então* existe uma expressão SQL representando cada uma dessas relações derivadas.

Etapa 3. Da junção das etapas 1 e 2 segue-se que qualquer relação derivável por meio de uma única expressão algébrica arbitrariamente complexa é também derivável por meio de uma expressão SQL adequada.

Vejamos agora os detalhes.

Etapa 1.

(Já feita).

Etapa 2

Suponhamos que as expressões SQL que criam A e B sejam, respectivamente,

```
SELECT column-list-A and SELECT column-list-B  
FROM   table-list-A      FROM   table-list-B  
WHERE  predicate-A        WHERE  predicate-B
```

Vamos nos referir a essas duas expressões como QA e QB. Vamos considerá-las posteriormente quando seja razoável supor que QA e QB existem.

<i>Algebra</i>	<i>SQL</i>
A UNION B	QA UNION QB
A MINUS B	QA AND NOT EXISTS (QB AND column-list-A = column-list-B)
A TIMES B	SELECT column-list-A,column-list-B FROM table-list-A,table-list-B WHERE predicate-A AND predicate-B
A WHERE p	SELECT column-list-A FROM table-list-A WHERE predicate-A AND p
A[x]	SELECT x FROM table-list-A WHERE predicate-A

Etapa 3

Das etapas 1 e 2 segue-se que *se* existe uma expressão SQL da forma assumida correspondente à aplicação de um único operador algébrico, *então* existe uma expressão SQL correspondente à aplicação de dois operadores algébricos em seqüência, e *consequentemente* existe uma expressão SQL para a seqüência de 3, 4, ..., qualquer quantidade de operadores.

A falha é que *não* existe uma expressão SQL da forma assumida (SELECT-FROM-WHERE) correspondente ao operador algébrico UNION. Portanto, na etapa 2 nossa suposição deveria ter sido que QA e QB fossem expressões SQL da forma

```
Q UNION Q UNION Q . . . . .
```

onde cada Q seria um bloco SELECT-FROM-WHERE. Fica como exercício para o leitor reexaminar as "equivalências" da Etapa 2 sob esta suposição revista e mostrar que a SQL é sem dúvida relationalmente completa no sentido mais abrangente do termo.

12.3 A armadilha é que a junção envolve os atributos de CITY bem como os atributos de S# e P#.

Resultado:

S.S#	S.SNAME	S.STATUS	S.CITY	SP.P#	SP.QTY	P.PNAME	P.COLOR	P.WEIGHT
S1	Smith	20	London	P1	300	Nut	Red	12
S1	Smith	20	London	P4	200	Screw	Red	14
S1	Smith	20	London	P6	100	Cog	Red	19
S2	Jones	10	Paris	P2	400	Bolt	Green	17
S3	Blake	30	Paris	P2	200	Bolt	Green	17
S4	Clark	20	London	P4	200	Screw	Red	14

12.4.1 J

12.4.2 J WHERE CITY='LONDON'

12.4.3 PX ALIASES P;

PY ALIASES P;

P[P#] MINUS

((PX[P#,WEIGHT] TIMES PY[WEIGHT])
WHERE PX.WEIGHT>PY.WEIGHT)[P#]

12.4.4 (SPJ WHERE J#='J1')[S#]

12.4.5 (SPJ WHERE J#='J1' AND P#='P1')[S#]

12.4.6 (J JOIN (SPJ WHERE S#='S1')[J#])[JNAME]

12.4.7 (P JOIN (SPJ WHERE S#='S1')[P#])[COLOR]

12.4.8 (SPJ WHERE J#='J1')[P#]

INTERSECT

(SPJ WHERE J#='J2')[P#]

12.4.9 ((SPJ WHERE J#='J1') JOIN (P WHERE COLOR='RED'))[S#]

12.4.10 (SPJ JOIN (J WHERE CITY='LONDON'))[P#]

12.4.11 ((J WHERE CITY='LONDON' OR CITY='PARIS')[J#]

JOIN SPJ

JOIN (P WHERE COLOR='RED')[P#])[S#]

12.4.12 (J[J#,CITY] JOIN SPJ JOIN S[S#,CITY])[P#]

12.4.13 ((J WHERE CITY='LONDON')[J#]

JOIN SPJ

JOIN (S WHERE CITY='LONDON')[S#])[P#]

12.4.14 ((J[J#,CITY] TIMES SPJ TIMES S[S#,CITY])

WHERE J.J#=SPJ.J# AND SPJ.S#=S.S#

AND J.CITY=S.CITY)[J#]

Observe que JUNÇÃO, como definimos, não tem utilidade aqui.

12.4.15 J[J#] MINUS

((S WHERE CITY='LONDON')[S#]

JOIN SPJ

JOIN (P WHERE COLOR='RED')[P#])[J#])

12.4.16 (((((SPJ JOIN (P WHERE COLOR='RED')[P#])[S#])

JOIN SPJ)[P#] JOIN SPJ)[S#])

12.4.17 ((SPJ WHERE S#='S1')[P#] JOIN SPJ)[J#]

12.4.18 ((S[S#,CITY] TIMES SPJ TIMES J[J#,CITY])
 WHERE S.S#=SPJ.S# AND SPJ.J#=J.J#)
 [S.CITY,J.CITY]
12.4.19 ((S[S#,CITY] TIMES SPJ TIMES J[J#,CITY])
 WHERE S.S#=SPJ.S# AND SPJ.J#=J.J#)
 [S.CITY,P#,J.CITY]
12.4.20 ((S[S#,CITY] TIMES SPJ TIMES J[J#,CITY])
 WHERE S.S#=SPJ.S# AND SPJ.J#=J.J#
 AND S.CITY=J.CITY)
 [S.CITY,P#,J.CITY]
12.4.21 (SPJ[S#,P#,J#] DIVIDE BY J[J#])[S#]
12.4.22 (SPJ WHERE S#='S1')[J#]
 MINUS
 (SPJ WHERE S#= 'S1')[J#]
12.4.23 SPJ[P#,J#] DIVIDE BY (J WHERE CITY='LONDON')[J#]
12.4.24 SPJ[J#,P#] DIVIDE BY (SPJ WHERE S#='S1')[P#]
12.4.25 J[J#] MINUS
 ((SPJ JOIN (P[P#] MINUS
 (SPJ WHERE S#='S1')[P#]))[J#])
12.4.26 SPJ[J#,S#,P#] DIVIDE BY (SPJ WHERE S#='S1')[S#,P#]
12.4.27 J[J#] MINUS
 ((SPJ[J#,P#] MINUS
 (SPJ WHERE S#='S1')[J#,P#]))[J#]
12.4.28 SPJ[J#,S#] DIVIDE BY (SPJ JOIN
 (P WHERE COLOR='RED')[P#])[S#]

CAPÍTULO 13: CÁLCULO RELACIONAL

13.1 (a) Verdadeira. (b) Verdadeira. (c) Falsa. (d) Verdadeira. Observe que os quantificadores de *semelhança* podem ser escritos em qualquer ordem, enquanto que para os quantificadores de *dessemelhança* a seqüência é significativa. Como uma ilustração, suponhamos que x e y são inteiros e que f é a fórmula " $y > x$ ". Deve ficar claro que $\forall x(\exists y(y > x))$ é verdadeiro, enquanto que $\exists y(\forall x(y > x))$ é falso. Portanto, o intercâmbio dos quantificadores de dessemelhança muda o significado da fórmula.

13.2 Para cada uma das questões a seguir daremos primeiro uma solução de cálculo de tupla, e depois uma solução de cálculo de domínio. As variáveis tupla e as variáveis domínio recebem os nomes do corpo do capítulo; não mostraremos suas declarações.

13.2.1 JX.J#,JX.JNAME,JX.CITY

```
JX,JNAME,X,CITYX WHERE J(J#:JX,  

    JNAME:JNAMEX,  

    CITY:CITYX)
```

Uma abreviação óbvia na solução do cálculo de tupla seria permitir simplesmente "JX" na lista alvo (como abreviatura para "JX.J#,JX.JNAME,JX.CITY").

13.2.2 JX.J#,JX.JNAME,JX.CITY WHERE JX.CITY='LONDON'

```
JX,JNAME,X,CITYX WHERE J(J#:JX,  

    JNAME:JNAMEX,  

    CITY:CITYX) AND  

    CITYX='LONDON'
```

13.2.3 PX.P# WHERE VPY(PY.WEIGHT>=PX.WEIGHT)
 PX WHERE VWEIGHTZ(IF ƎWEIGHTX(P(P#:PX,WEIGHT:WEIGHTX) AND
 P(WEIGHT:WEIGHTZ))
 THEN WEIGHTZ>=WEIGHTX)

13.2.4 SPJX.S# WHERE SPJX.J#='J1'
 SX WHERE SPJ(S#:SX,J#:'J1')

13.2.5 SPJX.S# WHERE SPJX.P#='P1' AND SPJX.J#='J1'
 SX WHERE SPJ(S#:SX,P#:'P1',J#:'J1')

13.2.6 JX.JNAME WHERE ƎSPJX(SPJX.J#=JX.J# AND
 SPJX.S#='S1')
 JNAMEX WHERE ƎJX(J(J#:JX,JNAME:JNAMEX) AND
 SPJ(J#:JX,S#:'S1'))

13.2.7 PX.COLOR WHERE ƎSPJX(SPJX.P#=PX.P# AND
 SPJX.S#='S1')
 COLORX WHERE ƎPX(P(P#:PX,COLOR:COLORX) AND
 SPJ(P#:PX,S#:'S1'))

13.2.8 SPJX.S# WHERE SPJX.J#='J1' AND
 ƎSPJY(SPJY.S#=SPJX.S# AND
 SPJY.J#= 'J2')
 SX WHERE SPJ(S#:SX,J#:'J1') AND
 SPJ(S#:SX,J#:'J2')

13.2.9 SPJX.S# WHERE ƎPX(PX.COLOR='RED' AND
 SPJX.P#=PX.P# AND
 SPJX.J#='J1')
 SX WHERE ƎPX(P(P#:PX,COLOR:'RED') AND
 SPJ(S#:SX,P#:PX,J#:'J1'))

13.2.10 SPJX.P# WHERE ƎJX(JX.CITY='LONDON' AND
 SPJX.J#=JX.J#)
 PX WHERE ƎJX(SPJ(P#:PX,J#:JX) AND
 J(J#:JX,CITY:'LONDON'))

13.2.11 SPJX.S# WHERE ƎPX(ƎJX(PX.COLOR='RED' AND
 C(JX.CITY='LONDON' OR
 JX.CITY='PARIS') AND
 SPJX.P#=PX.P# AND
 SPJX.J#=JX.J#))
 SX WHERE ƎPX(ƎJX(SPJ(S#:SX,P#:PX,J#:JX) AND
 P(P#:PX,COLOR:'RED') AND
 (J(J#:JX,CITY:'LONDON') OR
 J(J#:JX,CITY:'PARIS'))))

13.2.12 SPJX.P# WHERE ƎSX(ƎJX(SX.CITY=JX.CITY AND
 SPJX.S#=SX.S# AND
 SPJX.J#=JX.J#))
 PX WHERE ƎSX(ƎJX(ƎCITYX(SPJ(S#:SX,P#:PX,J#:JX) AND
 S(S#:SX,CITY:CITYX) AND
 J(J#:JX,CITY:CITYX)))

13.2.13 SPJX.P# WHERE \exists SX(\exists JX(SX.CITY='LONDON' AND
 JX.CITY='LONDON' AND
 SX.S#=SPJX.S# AND
 JX.J#=SPJX.J#))
 PX WHERE \exists SX(\exists JX(SPJ(S#:SX,P#:PX,J#:JX) AND
 S(S#:SX,CITY:'LONDON') AND
 J(J#:JX,CITY:'LONDON')))

13.2.14 SPJX.P# WHERE \exists SX(\exists JX(SX.CITY=JX.CITY AND
 SPJX.S#=SX.S# AND
 SPJX.J#=JX.J#))
 PX WHERE \exists SX(\exists JX(\exists CITYX(\exists CITYZ(SPJ(S#:SX,P#:PX,J#:JX) AND
 S(S#:SX,CITY:CITYX) AND
 J(J#:JX,CITY:CITYZ) AND
 CITYX=CITYZ)))

13.2.15 JX.J# WHERE NOT \exists SPJX(\exists PX(SX.CITY='LONDON' AND
 PX.COLOR='RED' AND
 SPJX.S#=SX.S# AND
 SPJX.P#=PX.P# AND
 SPJX.J#=JX.J#))
 JX WHERE J(J#:JX) AND
 NOT \exists PX(SPJ(S#:SX,P#:PX,J#:JX) AND
 S(S#:SX,CITY:'LONDON') AND
 P(P#:PX,COLOR:'RED'))

13.2.16 SPJX.S# WHERE \exists SPJY(SPJY.P#=SPJX.P# AND
 \exists SPJZ(SPJZ.S#=SPJY.S# AND
 \exists PX(PX.P#=SPJZ.P# AND
 PX.COLOR='RED')))
 SX WHERE \exists PX(\exists SY(\exists PY(SPJ(S#:SX,P#:PX) AND
 SPJ(P#:PX,S#:SY) AND
 SPJ(S#:SY,P#:PY) AND
 P(P#:PY,COLOR:'RED'))))

13.2.17 SPJX.J# WHERE \exists SPJY(SPJX.P#=SPJY.P# AND
 SPJY.S#='S1')
 JX WHERE \exists PX(SPJ(J#:JX,P#:PX) AND
 SPJ(P#:PX,S#:'S1'))

13.2.18 SX.CITY,JX.CITY WHERE \exists SPJX(SPJX.S#=SX.S# AND
 SPJX.J#=JX.J#)
 CITYX,CITYZ WHERE \exists SX(\exists JZ(S(S#:SX,CITY:CITYX) AND
 J(J#:JZ,CITY:CITYZ) AND
 SPJ(S#:SX,J#:JZ)))

13.2.19 SX.CITY,SPJX.P#,JX.CITY WHERE \exists SPJY(SPJY.S#=SX.S# AND
 SPJY.P#=SPJX.P# AND
 SPJY.J#=JX.J#)
 CITYX,PY,CITYZ WHERE \exists SX(\exists JZ(S(S#:SX,CITY:CITYX) AND
 J(J#:JZ,CITY:CITYZ) AND
 SPJ(S#:SX,P#:PY,J#:JZ)))

13.2.20 SX.CITY,SPJX.P#,JX.CITY WHERE \exists SPJY(SPJY.S#=SX.S# AND
 SPJY.P#=SPJX.P# AND
 SPJY.J#=JX.J# AND
 SX.CITY \sqsubset =JX.CITY)

CITYX,PY,CITYZ WHERE \exists SX(\exists JZ(S(S#:SX,CITY:CITYX) AND
 J(J#:JZ,CITY:CITYZ) AND
 SPJ(S#:SX,P#:PY,J#:JZ) AND
 CITYX \sqsubset =CITYZ))

13.2.21 SPJX.S# WHERE \forall JX(\exists SPJY(SPJY.S#=SPJX.S# AND
 SPJY.P#=SPJX.P# AND
 SPJY.J#=JX.J#))

SX WHERE \exists PX(\forall JX(SPJ(S#:SX,P#:PX,J#:JX)))

13.2.22 SPJX.J# WHERE \forall SPJY(IF SPJY.J#=SPJX.J#
 THEN SPJY.S#='S1')

JX WHERE \forall SX(IF SPJ(S#:SX,J#:JX)
 THEN SX='S1')

13.2.23 SPJX.P# WHERE \forall JX(IF JX.CITY='LONDON'
 THEN \exists SPJY(SPJY.P#=SPJX.P# AND
 SPJY.J#=JX.J#))

PX WHERE \forall JX(IF J(J#:JX,CITY:'LONDON')
 THEN SPJ(P#:PX,J#:JX))

13.2.24 SPJX.J# WHERE \forall SPJY(IF SPJY.S#='S1'
 THEN \exists SPJZ(SPJZ.J#=SPJX.J# AND
 SPJZ.P#=SPJY.P#))

JX WHERE \forall PX(IF SPJ(S#:'S1',P#:PX)
 THEN SPJ(P#:PX,J#:JX))

13.2.25 SPJX.J# WHERE \forall SPJY(IF SPJY.J#=SPJX.J#
 THEN \exists SPJZ(SPJZ.P#=SPJY.P# AND
 SPJZ.S#='S1'))

JX WHERE \forall PX(IF SPJ(J#:JX,P#:PX)
 THEN SPJ(P#:PX,S#:'S1'))

13.2.26 SPJX.J# WHERE \forall SPJY(IF SPJY.S#='S1'
 THEN \exists SPJZ(SPJZ.S#='S1' AND
 SPJZ.P#=SPJY.P# AND
 SPJZ.J#=SPJX.J#))

JX WHERE \forall PX(IF SPJ(S#:'S1',P#:PX)
 THEN SPJ(S#:'S1',P#:PX,J#:JX))

13.2.27 SPJX.J# WHERE \forall SPJY(IF SPJY.J#=SPJX.J#
 THEN \exists SPJZ(SPJZ.P#=SPJY.P# AND
 SPJZ.J#=SPJX.J# AND
 SPJZ.S#='S1'))

JX WHERE \forall PX(IF SPJ(P#:PX,J#:JX)
 THEN SPJ(P#:PX,J#:JX,S#:'S1'))

13.2.28 SPJX.J# WHERE \forall SPJY(IF \exists PX(PX.COLOR='RED' AND
 PX.P#=SPJY.P#)
 THEN \exists SPJZ(SPJZ.S#=SPJY.S# AND
 SPJZ.J#=SPJZ.J#))

```
JX WHERE VSX(IF SPX(SPJ(S#:SX,P#:PX) AND  
P(P#:PX,COLOR:'RED'))  
THEN SPJ(S#:SX,J#:JX))
```

13.3 Novamente daremos primeiro as soluções de cálculo de tupla. Serão omitidas as soluções que não diferirem da solução correspondente sob 13.2.

13.3.6 JX.JNAME WHERE JX.J#=SPJX.J# AND
SPJX.S#='S1'

```
JNAMEX WHERE JC(J#:JX,JNAME:JNAMEX) AND  
SPJ(J#:JX,S#:'S1')
```

13.3.7 PX.COLOR WHERE PX.P#=SPJX.P# AND
SPJX.S#='S1'

```
COLORX WHERE P(P#:PX,COLOR:COLORX) AND  
SPJ(P#:PX,S#:'S1')
```

13.3.8 SPJX.S# WHERE SPJX.J#='J1' AND
SPJX.S#=SPJY.S# AND
SPJY.J#='J2'

```
SX WHERE SPJ(S#:SX,J#:'J1') AND  
SPJ(S#:SX,J#:'J2')
```

13.3.9 SPJX.S# WHERE SPJX.J#='J1' AND
SPJX.P#=PX.P# AND
PX.COLOR='RED'

```
SX WHERE P(P#:PX,COLOR:'RED') AND  
SPJ(S#:SX,P#:PX,J#:'J1')
```

13.3.10 SPJX.P# WHERE SPJX.J#=JX.J# AND
JX.CITY='LONDON'

```
PX WHERE SPJ(P#:PX,J#:JX) AND  
J(J#:JX,CITY:'LONDON')
```

13.3.11 SPJX.S# WHERE SPJX.P#=PX.P# AND
SPJX.J#=JX.J# AND
PX.COLOR='RED' AND
(JX.CITY='LONDON' OR
JX.CITY='PARIS')

```
SX WHERE SPJ(S#:SX,P#:PX,J#:JX) AND  
P(P#:PX,COLOR:'RED') AND  
(J(J#:JX,CITY:'LONDON') OR  
J(J#:JX,CITY:'PARIS'))
```

13.3.12 SPJX.P# WHERE SPJX.S#=SX.S# AND
SPJX.J#=JX.J# AND
SX.CITY=JX.CITY

```
PX WHERE SPJ(S#:SX,P#:PX,J#:JX) AND  
S(S#:SX,CITY:CITYX) AND  
J(J#:JX,CITY:CITYX)
```

13.3.13 SPJX.P# WHERE SPJX.S#=SX.S# AND
SPJX.J#=JX.J# AND
SX.CITY='LONDON' AND
JX.CITY='LONDON'

PX WHERE SPJ(S#:SX,P#:PX,J#:JX) AND
S(S#:SX,CITY:'LONDON') AND
J(J#:JX,CITY:'LONDON')

13.3.14 SPJX.P# WHERE SPJX.S#=SX.S# AND
SPJX.J#=JX.J# AND
SX.CITY=JX.CITY

PX WHERE SPJ(S#:SX,P#:PX,J#:JX) AND
S(S#:SX,CITY:CITYX) AND
J(J#:JX,CITY:CITYZ) AND
CITYX=CITYZ

13.3.16 SPJX.S# WHERE SPJX.P#=SPJY.P# AND
SPJY.S#=SPJZ.S# AND
SPJZ.P#=PX.P# AND
PX.COLOR='RED'

SX WHERE SPJ(S#:SX,P#:PX) AND
SPJ(P#:PX,S#:SY) AND
SPJ(S#:SY,P#:PY) AND
P(P#:PY,COLOR:'RED')

13.3.17 SPJX.J# WHERE SPJX.P#=SPJY.P# AND
SPJY.S#=S1

JX WHERE SPJ(J#:JX,P#:PX) AND
SPJ(P#:PX,S#:S1)

13.3.18 SX.CITY,JX.CITY WHERE SX.S#=SPJX.S# AND
SPJX.J#=JX.J#

CITYX,CITYZ WHERE S(S#:SX,CITY:CITYX) AND
J(J#:JZ,CITY:CITYZ) AND
SPJ(S#:SX,J#:JZ)

13.3.19 SX.CITY,SPJX.P#,JX.CITY WHERE SX.S#=SPJY.S# AND
SPJX.P#=SPJY.P# AND
JX.J#=SPJY.J#

CITYX,PY,CITYZ WHERE S(S#:SX,CITY:CITYX) AND
J(J#:JZ,CITY:CITYZ) AND
SPJ(S#:SX,P#:PY,J#:JZ)

13.3.20 SX.CITY,SPJX.P#,JX.CITY WHERE SX.S#=SPJY.S# AND
SPJX.P#=SPJY.P# AND
JX.J#=SPJY.J# AND
SX.CITY=JX.CITY

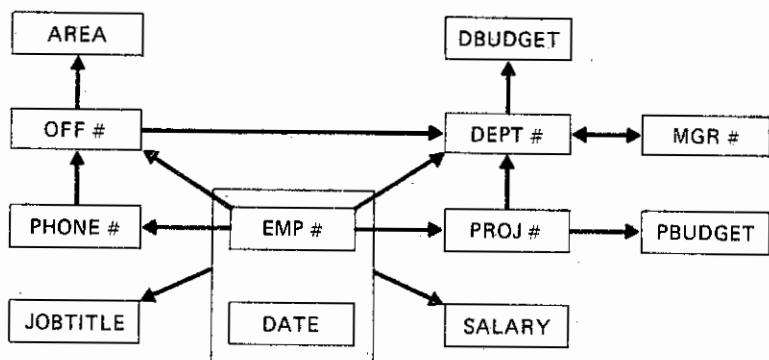
CITYX,PY,CITYZ WHERE S(S#:SX,CITY:CITYX) AND
J(J#:JZ,CITY:CITYZ) AND
SPJ(S#:SX,P#:PY,J#:JZ) AND
CITYX=CITYZ

13.3.21 SPJX.S# WHERE VJX(ESPJY(SPJY.S#=SPJX.S# AND
SPJY.P#=SPJX.P# AND
SPJY.J#=JX.J#))

SX WHERE VJX(SPJ(S#:SX,P#:PX,J#:JX))

CAPÍTULO 14: NORMALIZAÇÃO ADICIONAL

14.1 O diagrama mostra todas as dependências funcionais diretas envolvidas, tanto as implícitas na enunciação do problema quanto as que correspondem a uma “*suposição razoável*” sobre a semântica (estabelecido explicitamente abaixo). Os nomes de atributos se propõem ser auto-explicativos.



Para dependências de *múltiplos valores*, veja a etapa 1 abaixo. Estamos supondo que não haja dependências de junção adicionais (isto é, JDs que não sejam FDs ou MVDs).

Suposições semânticas

- Nenhum empregado é gerente de mais de um departamento ao mesmo tempo.
- Nenhum empregado trabalha em mais de um departamento ao mesmo tempo.
- Nenhum empregado trabalha em mais de um projeto ao mesmo tempo.
- Nenhum empregado tem mais do que um escritório ao mesmo tempo.
- Nenhum empregado tem mais do que um telefone ao mesmo tempo.
- Nenhum empregado tem mais do que um trabalho ao mesmo tempo.
- Nenhum projeto é designado para mais de um departamento ao mesmo tempo.
- Nenhum escritório é designado para mais do que um departamento ao mesmo tempo.

Etapa 0

Primeiramente observe que a estrutura hierárquica pode ser considerada como uma relação não normalizada DEPT0 definida nos domínios DEPT# (a chave primária), DBUDGET, MGR#, e três domínios adicionais cujos elementos são não-atômicos: XEMP0, XPROJ0, e XOFFICE0, digamos. Podemos representar esta relação não-normalizada como

DEPT0 (DEPT#, DBUDGET, MGR#, XEMP0, XPROJ0, XOFFICE0)

(a chave primária está indicada pelo sublinhado). Vamos ignorar empregados e escritórios por um momento, e vamos nos concentrar nos projetos. Definamos a relação

PROJ0 (PROJ#, PBUDGET)

como sendo o conjunto de *todos* os pares PROJ#-PBUDGET. Então o valor de XPROJ0 associado a algum departamento em particular é algum subconjunto deste conjunto. Portanto, o domínio XPROJ0 consiste do conjunto de todos os subconjuntos de PROJ0 (o chamado conjunto potência de PROJ0).

Observações semelhantes aplicam-se a XEMPO, XOFFICE0, e a todos os domínios do exemplo cujos elementos são não-atômicos; em cada caso o domínio é o conjunto potência de uma relação definida como o conjunto de todas as tuplas do tipo particular. Vamos indicar cada conjunto potência por um prefixo X. Então, a coleção completa de relações, normalizadas e não-normalizadas, é a que se segue.

```
DEPTO(DEPT#, DBUDGET, MGR#, XEMPO, XPROJO, XOFFICE0)
EMP0(EMP#, PROJ#, OFF#, PHONE#, XJOB0)
JOB0(JOBTITLE, XSALHIST0)
SALHIST0(DATE, SALARY)
PROJ0(PROJ#, PBUDGET)
OFFICE0(OFF#, AREA, XPHONE0)
PHONE0(PHONE#)
```

Etapa 1

Vamos agora reduzir este conjunto a uma coleção de relações 1NF. O processo preliminar de redução é explicado por Codd [4.1] como a seguir. Começando pela relação no topo da hierarquia, tomamos sua chave primária e expandimos cada uma das relações imediatamente subordinadas para inserir esta chave primária. A chave primária de cada relação expandida é a combinação da chave primária anterior à expansão juntamente com a chave primária copiada da relação pai. Agora apagamos da relação pai todos os atributos não-simples (isto é, aqueles cujos elementos são não-atômicos), removemos o nó do topo da hierarquia, e repetimos a mesma seqüência de operações em cada sub-hierarquia remanescente. Obteremos a seguinte coleção de relações 1NF. Observe que perdemos os conjuntos potência. De fato, considerando cada sub-hierarquia separadamente, eliminamos imediatamente todas as dependências de múltiplos valores que não são também dependências funcionais.

```
DEPT1(DEPT#, DBUDGET, MGR#)
EMP1(DEPT#, EMP#, PROJ#, OFF#, PHONE#)
JOB1(DEPT#, EMP#, JOBTITLE)
SALHIST1(DEPT#, EMP#, JOBTITLE, DATE, SALARY)
PROJ1(DEPT#, PROJ#, PBUDGET)
OFFICE1(DEPT#, OFF#, AREA)
PHONE1(DEPT#, OFF#, PHONE#)
```

Etapa 2

Podemos agora reduzir as relações 1NF a uma coleção equivalente 2NF eliminando as dependências não-totais. Vamos considerar as relações 1NF uma por uma.

DEPT1: Esta relação já está em 2NF.

EMP1: Observe primeiro que DEPT# é redundante como componente da chave primária desta relação. Podemos tomar EMP# sozinho como chave primária, caso em que a relação estará em 2NF.

JOB1: Novamente, observe que DEPT# não é requerido como componente da chave. Como DEPT# é funcionalmente dependente de EMP#, temos um atributo não-chave (DEPT#) que não é totalmente funcionalmente dependente da chave primária (a combinação EMP#-JOBTITLE), e por isso JOB1 não está em 2NF. Podemos substituí-la por

JOB2(EMP#, JOBTITLE)

e

JOB2'(EMP#, DEPT#)

Entretanto, JOB2 é uma projeção de SALHIST2 (veja abaixo), e JOB2' é uma projeção de EMP1 (renomeado como EMP2 abaixo); portanto essas duas relações podem ser descartadas.

SALHIST1: Como em JOB1, podemos projetar DEPT# inteiramente. Além disso, JOBTITLE não é requerido como um componente da chave; podemos tomar a combinação EMP#-DATE como chave primária, para obtermos a relação 2NF.

SALHIST2(EMP#, DATE, JOBTITLE, SALARY)

PROJ1: Como em EMP1, podemos considerar DEPT# como um atributo não-chave; a relação será então 2NF.

OFFICE1: Aplicam-se as mesmas observações.

PHONE1: Podemos projetar DEPT# inteiramente, pois a relação (DEPT#, OFF#) é uma projeção de OFFICE1 (renomeada OFFICE2 abaixo). Também, OFF# é funcionalmente dependente de PHONE#, e portanto podemos tomar PHONE# como chave primária, para obtermos a relação 2NF

PHONE2(PHONE#, OFF#)

Observe que isto não é necessariamente uma projeção de EMP2 (telefones ou escritórios podem existir sem que tenham sido designados a empregados), e assim não podemos descartar esta relação.

Portanto, nossa coleção de relações 2NF é

```
DEPT2(DEPT#, DBUDGET, MGR#)
EMP2(EMP#, DEPT#, PROJ#, OFF#, PHONE#)
SALHIST2(EMP#, DATE, JOBTITLE, SALARY)
PROJ2(PROJ#, DEPT#, PBUDGET)
OFFICE2(OFF#, DEPT#, AREA)
PHONE2(PHONE#, OFF#)
```

Etapa 3

Agora podemos reduzir as relações 2NF a um conjunto equivalente 3NF eliminando as dependências transitivas. A única relação 2NF que já não é 3NF é a relação EMP2, na qual OFF# e DEPT# são ambos transitivamente dependentes da chave primária EMP#: OFF# via PHONE#, e DEPT# via PROJ# e via OFF# (e consequentemente PHONE#). As relações 3NF (projeções) correspondentes a EMP2 são

```
EMP3(EMP#, PROJ#, PHONE#)
X(PHONE#, OFF#)
Y(PROJ#, DEPT#)
Z(OFF#, DEPT#)
```

Entretanto, X é PHONE2, Y é uma projeção de PROJ2, e Z é uma projeção de OFFICE2. Portanto nossa coleção de relações 3NF é

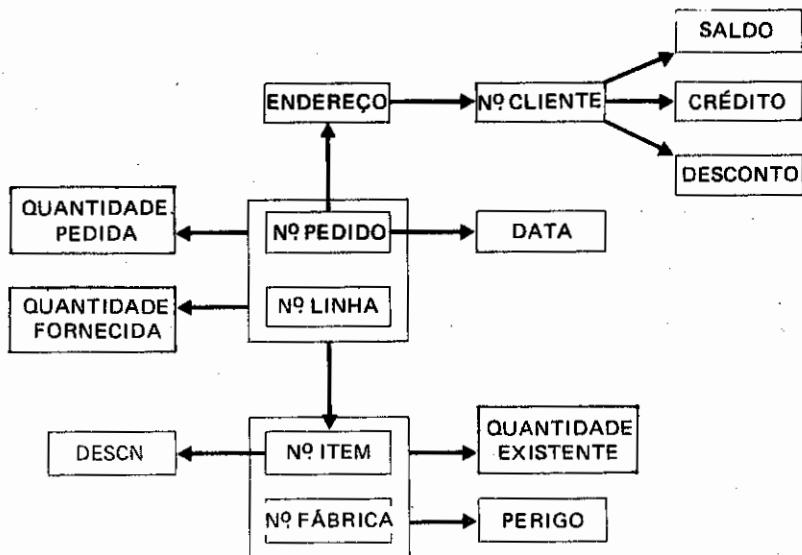
```
DEPT3(DEPT#, DBUDGET, MGR#)
EMP3(EMP#, PROJ#, PHONE#)
SALHIST3(EMP#, DATE, JOBTITLE, SALARY)
PROJ3(PROJ#, DEPT#, PBUDGET)
OFFICE3(OFF#, DEPT#, AREA)
PHONE3(PHONE#, OFF#)
```

Cada uma dessas relações 3NF é de fato BCNF, e sem dúvida 4NF (deixá-la à maneira como nós executamos a redução para 1NF na etapa 1). Pela nossa suposição no que se refere a JDs, elas são também 5NF, e portanto a decomposição está completa. Observe que em DEPT3 nós temos duas chaves candidatas, DEPT# e MGR#.

Observemos também que, dadas certas restrições semânticas adicionais (razoáveis), esta coleção de relações é **fortemente redundante** [4.1], pois a projeção da relação PROJ3 sobre (PROJ#, DEPT#) é uma projeção da junção de EMP3 e PHONE3 e OFFICE3.

Observemos finalmente que é possível "descobrirmos" as relações 3NF a partir do diagrama de dependência funcional. (Como?)

14.2 O diagrama mostra todas as dependências funcionais diretas envolvidas.



Suposições semânticas

- Não há dois clientes com o mesmo endereço de remessa.
- Cada pedido é identificado por um número único de pedido.
- Cada linha de detalhe dentro de um pedido é identificada por um número de linha, único dentro do pedido.

Relações 4NF (5NF)

```

CUST(CUST#, BAL, CREDLIM, DISCOUNT)
SHIPTO(ADDRESS, CUST#)
ORDHEAD(ORD#, ADDRESS, DATE)
ORDLINE(ORD#, LINE#, ITEM#, QTYORD, QTYOUT)
ITEM(ITEM#, DESCN)
IP(ITEM#, PLANT#, QTYOH, DANGER)
    
```

14.3 Consideremos o processamento que tem que ser executado por um programa que manuseie os pedidos. Estamos supondo que o pedido entrado especifica o número do cliente, o endereço para remessa e detalhes dos itens pedidos (números e quantidades dos itens).

```

SELECT * FROM CUST WHERE CUST.CUST#=input.CUST#
check balance, credit limit etc
SELECT * FROM SHIPTO WHERE SHIPTO.ADDR=input.ADDR AND
SHIPTO.CUST#=input.CUST#
    
```

(isto verifica o endereço de remessa
se tudo estiver OK prossiga e processe o pedido)

Se 99 por cento dos clientes têm apenas um endereço para remessa, seria bastante ineficiente colocar-se esse endereço em uma relação outra que não CUST (considerando somente os 99 por cento, ADDR é de fato funcionalmente dependente de CUST#). Podemos trazer uma melhoria como se segue. Para

cada cliente, designamos um endereço válido de remessa como sendo o endereço *primário* do cliente. Naturalmente, para os 99 por cento, o endereço primário será o único. Qualquer endereço adicional trataremos como *secundário*. A relação CUST pode então ser redefinida como

CUST(CUST#, ADDR, BAL, CREDLIM, DISCOUNT)

e a relação SHIPTO pode ser substituída por

SECOND(ADDR, CUST#)

Aqui CUST.ADDR refere-se ao endereço primário, e SECOND contém todos os endereços secundários (e os números de clientes correspondentes). Essas relações são 4NF. O programa de processamento de pedidos agora se parecerá com o seguinte:

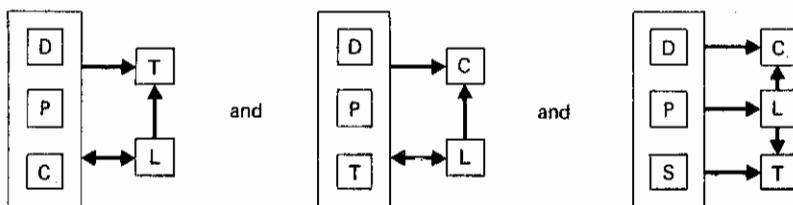
```
SELECT * FROM CUST WHERE CUST.CUST#=input.CUST#
check balance, credit limit etc
IF CUST.ADDR#=input.ADDR THEN
  SELECT * FROM SECOND WHERE SECOND.ADDR=input.ADDR AND
  SECOND.CUST#=input.CUST#
  (isto verifica o endereço de remessa
  se tudo estiver OK prossiga e processe o pedido)
```

As vantagens desta abordagem são as seguintes:

- O processamento é mais simples e marginalmente mais eficiente para 99 por cento dos clientes.
- Se o endereço de remessa for omitido na entrada do pedido, o endereço primário pode ser usado por *default*.
- Suponhamos que o cliente possa ter descontos diferentes para cada endereço de remessa. Na abordagem original (mostrada na resposta ao Exercício 14.2), o atributo DISCOUNT teria que ser movido para a relação SHIPTO, tornando o processamento ainda mais complicado. Na abordagem revisada, entretanto, o desconto primário (correspondente ao endereço primário) pode ser representado por um aparecimento de DISCOUNT em CUST, e os descontos secundários por um aparecimento correspondente de DISCOUNT em SECOND. Ambas as relações estão ainda em 4NF, e o processamento é novamente mais simples para 99 por cento dos clientes.

Resumindo: O isolamento de casos excepcionais é provavelmente uma técnica valiosa para se obter o melhor dos dois mundos – isto é, combinar as vantagens do 4NF com a simplificação nas operações de recuperação que podem ocorrer se as restrições ao 4NF forem violadas.

14.4 O diagrama ilustra as dependências funcionais (mais importantes).



Uma coleção possível de relações 4NF é

```
RELATION SCHED (L,T,C,D,P)
  PRIMARY KEY (L)
  ALTERNATE KEY (T,D,P)
  ALTERNATE KEY (C,D,P)

RELATION STUDY (S,L)
  PRIMARY KEY (S,L)
```

Esta redução não é única.

14.5 (a) Verdadeiro.

(b) Verdadeiro.

(c) Falso (mas "quase verdadeiro"). Uma relação binária $R(X, Y)$ pode ser decomposta sem perdas em suas duas projeções unárias $R1(X)$, $R2(Y)$ se e somente se R for igual ao produto Cartesiano de $R1$ e $R2$ (lembre-se de que junção e produto Cartesiano são idênticos se as relações sendo juntadas não tiverem atributo comum). Veja a Fig. 4.2 para um exemplo. Essa relação satisfaz à especial, não-trivial $MVD\emptyset \rightarrow\rightarrow X \sqcap Y$ (onde \emptyset é o conjunto vazio), e portanto não é 4NF.

(d) Falso; veja (c).

(e) Falso (verdadeiro se " $A \rightarrow B$ " for mudado para " $A \rightarrow\rightarrow B$ "). A parte "se" é verdadeira; exemplo que se contrapõe à parte "somente se":

R	A	B	C
a1	b1	c1	
a1	b2	c1	
a1	b1	c2	
a1	b2	c2	

R1	A	B
a1	b1	
a1	b2	

R2	A	C
a1	c1	
a1	c2	

R é a junção de R1 e R2, embora $A \rightarrow B$ não valha em R. (Nem $A \rightarrow C$ vale).

(f) Verdadeiro.

(g) Verdadeiro.

(h) Verdadeiro.

14.6 Introduziremos primeiramente três relações

REPS (REP#, . . .)
AREAS (AREA#, . . .)
PRODUCTS (PROD#, . . .)

de interpretação óbvia. Segundo, podemos representar a associação entre representantes de vendas e áreas de vendas por uma relação

RA (REP#, AREA#)

e a associação entre representantes de vendas e produtos por uma relação

RP (REP#, PROD#)

(ambas são associações de muitos-para-muitos).

A seguir, sabemos que todos os produtos são vendidos em todas as áreas. Portanto, introduziremos a relação

AP (AREA#, PROD#)

Para representar a associação entre áreas e produtos, e depois temos a restrição (C)

$\forall AX \ \forall PX \ \exists APX \ (APX.AREA\# = AX.AREA\# \text{ AND } APX.PROD\# = PX.PROD\#)$

(onde AX, PX, APX são variáveis tupla das relações AREAS, PRODUCTS, AP, respectivamente). Observe que a restrição C implica que a relação AP não seja 4NF (veja o Exercício 14.5).

Não há dois representantes vendendo o mesmo produto na mesma área. Em outras palavras, dada uma combinação (AREA#, PROD#), existe exatamente um representante de vendas responsável (REP#), e podemos então introduzir a relação

APR (AREA#, PROD#, REP#)

na qual (para tornar a dependência funcional explícita)

APR.(AREA#, PROD#) → APR.REP#

(Naturalmente, a especificação da combinação AREA#-PROD# como chave primária é suficiente para expressar esta FD.) Agora, entretanto, as relações RA, RP, e AP são todas redundantes, pois são todas projeções de APR; podem portanto ser retiradas. No lugar da restrição C, podemos agora precisar da restrição C1

**∀AX ∀PX ∃APRX (APRX.AREA# = AX.AREA# AND
APRX.PROD# = PX.PROD#)**

(onde APRX é uma variável tupla da relação APR). Esta restrição precisa certamente ser explícita, pois deve ser exigida pelo DBMS, mas teria que ser explicitada de qualquer maneira, pois representa parte da semântica da situação, precisando ser entendida pelo usuário. Além disso, como qualquer representante vende todos os seus produtos em sua área, temos a restrição C2.

APR.REP# → APR.AREA# | APR.PROD#

(uma dependência de múltiplos valores; a relação APR não está em 4 NF). Novamente, a restrição deve ser explicitada.

Portanto, o projeto final consiste das relações REPS, AREAS, PRODUCTS, e APR, juntamente com as constantes explícitas C1 e C2. Este exercício ilustra muito claramente o ponto de que, em geral, a disciplina de normalização é adequada para representar *alguns* aspectos semânticos de um determinado problema (FDs e MVDs que são consequência de chaves), mas a colocação explícita de FDs e MVDs adicionais pode também ser necessária por outros aspectos, sendo que alguns aspectos não podem ser representados de nenhuma forma por MVDs. Também ilustra o ponto de que nem sempre é desejável uma normalização "até o fim" (a relação APR está em BCNF mas não em 4NF).

Observe que são também necessárias restrições explícitas adicionais, além das discutidas. Por exemplo, precisamos da restrição

∀APRX ∃AX (AX.AREA# = APRX.AREA#)

para expressar o fato de que cada área mencionada em APR tem que existir em AREAS; e de forma semelhante para produtos e representantes. São também necessárias restrições semelhantes nos exercícios anteriores deste capítulo. Uma discussão abrangente sobre restrições está além do escopo deste livro.

CAPÍTULO 16: ESTRUTURA DE DADOS IMS

16.1 COURSE M23

PREREQ M16

PREREQ M19

OFFERING 730813

TEACHER 421633

STUDENT 102141

STUDENT 183009

STUDENT 761620

OFFERING 741104

OFFERING 750106

16.2 DBD NAME=PUBDBD

SEGMENT NAME=SUB, BYTES=45

FIELD NAME=(SUB#, SEQ), BYTES=7, START=1

FIELD NAME=SUBNAME, BYTES=38, START=8

```

SEGMENT NAME=PUB, PARENT=SUB, BYTES=45
FIELD NAME=(PUBNAME, SEQ, M), BYTES=44, START=1
FIELD NAME=AMFLAG, BYTES=1, START=45
SEGMENT NAME=DETAILS, PARENT=PUB, BYTES=25
FIELD NAME=(DATE, SEQ, M), BYTES=6, START=1
FIELD NAME=PUBHOUSE, BYTES=19, START=7
FIELD NAME=JVNVOOLISS, BYTES=19, START=7
SEGMENT NAME=AUTHOR, PARENT=PUB, BYTES=50
FIELD NAME=(AUTHNAME, SEQ), BYTES=16, START=1
FIELD NAME=AUTHADDR, BYTES=34, START=17

```

Observe as especificações M de PUBNAME e DATE.

CAPÍTULO 17: O NÍVEL EXTERNO DO IMS

```

17.1 PCB      TYPE=DB, DBDNAME=PUBDBD, KEYLEN=67
SENSEG NAME=SUB, PROCOPT=G
SENSEG NAME=PUB, PARENT=SUB, PROCOPT=G
SENSEG NAME=DETAILS, PARENT=PUB, PROCOPT=G
SENSEG NAME=AUTHOR, PARENT=PUB, PROCOPT=G

```

CAPÍTULO 18: MANIPULAÇÃO DE DADOS IMS

```

18.1 GU SUB(SUBNAME='INFORMATION RETRIEVAL')
GNP GNP AUTHOR
add author name to result list (eliminating duplicates)
go to GNP
18.2 GU SUB
GN GN PUB*D
    AUTHOR(AUTHNAME='GRACE')|AUTHNAME='HOBBS')
add publication name to result list
go to GN

```

Observe que o “or” está representado por uma | (barra vertical). O “and” é representado por um &. Com esta codificação, uma publicação aparecerá duas vezes na lista se Grace e Hobbs forem ambos autores daquela publicação.

```

18.3 GU SUB
GN GN SUB*D
    PUB(AMFLAG='M')
    AUTHOR(AUTHNAME='BRADBURY')
add subject name to result list (eliminating duplicates)
go to GN

```

Observe que poderia ter sido mais eficiente emitir outro GN SUB antes de desviar de volta. Observe também que se não nos interessasse saber se uma publicação é artigo ou monografia, poderíamos omitir a segunda SSA; o IMS assumiria uma SSA incondicional em PUB.

```

18.4 GU SUB
GN GN PUB(AMFLAG='A')
GNP AUTHOR(AUTHNAME='OWEN')
if not found go to GN
GNP DETAILS*F
add publication name and date to result list
go to GN

```

```

18.5 GU SUB
GN GN PUB(AMFLAG='M')
GNP DETAILS(PUBHOUSE='CIDER PRESS'&DATE>'700101')
if not found go to GN

```

GNP GNP AUTHOR
if not found go to GN
add author name to result list (eliminating duplicates)
go to GNP

18.6 build PUB and DETAILS segments concatenated in I/O area
ISRT SUB(SUBNAME='SCIENCE FICTION')
PUB*D
.DETAILS
build AUTHOR segment in I/O area
ISRT AUTHOR

18.7 GU SUB
GN GN PUB
set counter=0
GNP GNP DETAILS
if not found go to TST
set counter=counter+1
go to GNP
TST if counter=1 go to GN
GHNP DETAILS*F
DLET
set counter=counter-1
go to TST

18.8 O uso de V* ao invés de GNP tem o efeito de reduzir a quantidade de chamadas de sub-rotina ao IMS, melhorando o desempenho, nas Questões 18.4 e 18.5. Na solução acima ao 18.4, podemos substituir GN e o primeiro GNP (e o "if not found go to GN") por um único GN

GN PUB(AMFLAG='A')
AUTHOR(AUTHNAME='OWEN')

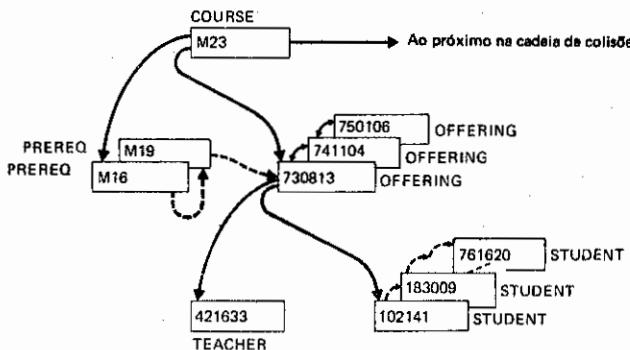
e o segundo GNP por

GN PUB*D
.DETAILS*F

(observe o uso de dois códigos de comando em um único SSA). As substituições no 18.5 são semelhantes, exceto que os códigos de comando D e F não são necessários).

CAPÍTULO 19: O NÍVEL INTERNO DO IMS

19.1



As setas tracejadas representam indicadores de localização hierárquicos; as setas cheias, indicadores de localização filho/gêmeo.

19.2

Segmento	Dado	Indicadores de localização (quantidade)	Tamanho do prefixo	Ocorrências por ocorrência PDBR (quantidade média)		Bytes de dados por ocorrência PDBR (quantidade)	Bytes de prefixo por ocorrência PDBR. (quantidade)
				PDBR	Bytes de dados por ocorrência PDBR (quantidade)		
COURSE	256	3	18	1	256	18	
PREREQ	36	1	10	2	72	20	
OFFERING	20	4	22	8	160	176	
TEACHER	24	1	10	12	288	120	
STUDENT	26	1	10	128	3328	1280	
					4104	1614	

(Foi adicionado um byte de preenchimento ao segmento STUDENT). Razão entre bytes de prefixo e de dados = $1614/4104 = 39\%$ aproximadamente.

Razão entre bytes de prefixo e total de bytes = $1614/5718 = 28\%$ aproximadamente.

19.3 Os casos (a) e (b) são impossíveis porque os indicadores de localização hierárquicos não podem cruzar os limites entre dois DSGs. O caso (c) pode ser especificado como a seguir

```

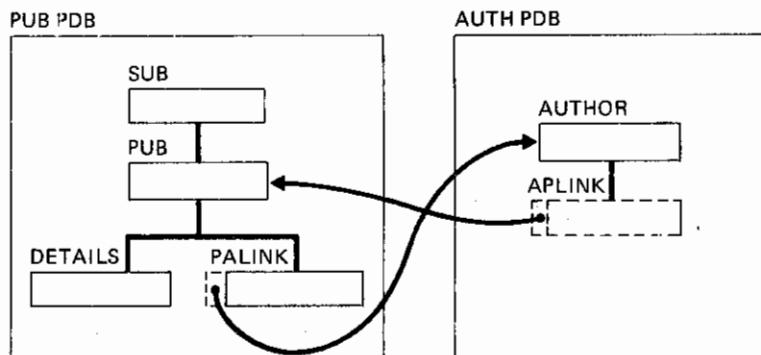
DBD . .
P DATASET . .
SEGMENT NAME=COURSE, . .
Q DATASET . .
SEGMENT NAME=PREREQ, . .
SEGMENT NAME=OFFERING, . .
P DATASET . .
SEGMENT NAME=TEACHER, . .
R DATASET . .
SEGMENT NAME=STUDENT, . .

```

O caso (a) é o único possível se a estrutura de armazenamento for HISAM (e portanto somente se o método de acesso de suporte for ISAM/OSAM).

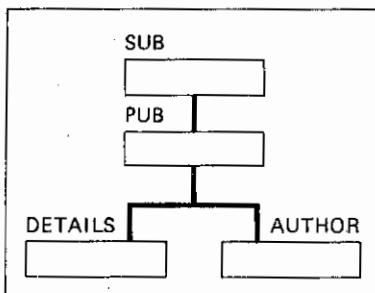
CAPÍTULO 20: BANCOS DE DADOS LÓGICOS DO IMS

20.1 O diagrama mostra um possível par de PDBs. Observe que PALINK e APLINK são segmentos parelhos, e que está sendo usada parelha virtual. APLINK foi escolhido (arbitrariamente) como membro virtual do par (na prática, essa escolha é feita com base em critérios de desempenho).

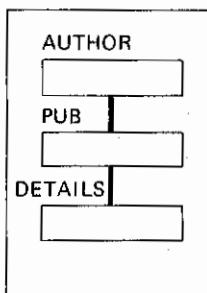


Podem ser definidos os seguintes LDBs. (Não estão mostrados os dados de interseção e as chaves totalmente concatenadas pai-lógico.)

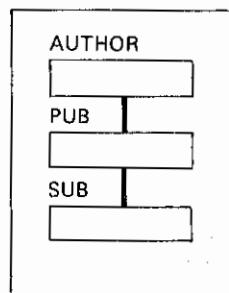
SPDA LDB



APD LDB



APS LDB



Os DBs físicos PUB e AUTH são essencialmente semelhantes aos DBDs físicos AREA e BIRD das Figs. 20.16 e 20.17. A diferença mais significativa está na especificação do campo SEQ de APLINK (o membro virtual do par):

```
SEGMENT NAME=APLINK, POINTER=PAIRED,  
        PARENT=AUTHOR, SOURCE=((PALINK,, PUBPDBD))  
FIELD NAME=(FCKEY, SEQ), START=1, BYTES=51  
FIELD NAME=SUB#, START=1, BYTES=7  
FIELD NAME=PUBNAME, START=8, BYTES=44
```

O campo de ordenação é a chave *totalmente concatenada* pai lógico, isto é, a combinação de SUB# e PUBNAME.

Os três DBDs lógicos são essencialmente similares ao DBD lógico da Fig. 20.9.

20.5 Para o EDUC PDB (Fig. 20.21):

```
DBD      NAME=EDUCPDBD, . . .  
SEGMENT NAME=COURSE, POINTER=TWIN, . . .  
LCHILD   NAME=(CP, EDUCPDBD), PAIR=PC, POINTER=SNGL  
FIELD    NAME=(COURSE#, SEQ), BYTES=3, START=1  
FIELD    NAME=TITLE, . . .  
FIELD    NAME=DESCRIPN, . . .  
SEGMENT NAME=CP, POINTER=(LPARNT, TWIN, LTWIN),  
        PARENT=((COURSE), (COURSE, PHYSICAL, EDUCPDBD))  
FIELD    NAME=(COURSE#, SEQ), . . .  
SEGMENT NAME=PC, POINTER=PAIRED, PARENT=COURSE,  
        SOURCE=((CP,, EDUCPDBD))  
FIELD    NAME=(COURSE#, SEQ), . . .
```

Para o EDCP LDB (Fig. 20.22):

```
DBD      NAME=EDCPLDBD, ACCESS=LOGICAL  
DATASET  LOGICAL  
SEGMENT NAME=COURSE, SOURCE=((COURSE,, EDUCPDBD))  
SEGMENT NAME=PREREQ, PARENT=COURSE,  
        SOURCE=((CP,, EDUCPDBD), (COURSE,, EDUCPDBD))
```

Para o EDPC LDB (Fig. 20.22):

```
DBD      NAME=EDPCLDBD, ACCESS=LOGICAL  
DATASET  LOGICAL
```

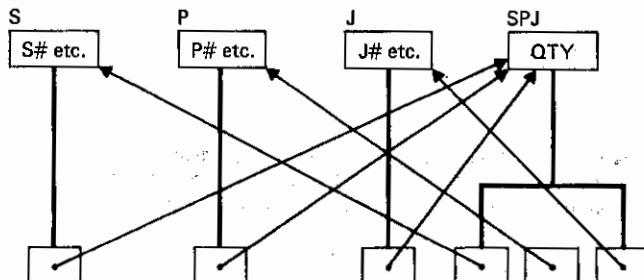
```

SEGM      NAME=PREREQ, SOURCE=((COURSE,,EDUCPDBD))
SEGM      NAME=COURSE, PARENT=PREREQ,
          SOURCE=((PC,,EDUCPDBD),(COURSE,,EDUCPDBD))

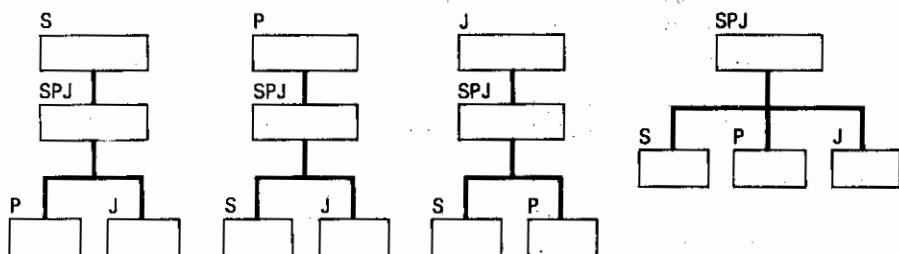
```

20.6 Se supusermos que os segmentos são apresentados para carga em seqüência hierárquica física, será geralmente verdadeiro que o pai lógico já existirá quando o filho lógico for submetido para inserção (embora seja sempre assim nos dados de exemplo do Exercício 20.3). Portanto, após a carga dos dados no banco de dados, será necessário correr um programa utilitário sobre ele para resolver todos os relacionamentos lógicos. Alternativamente, o processo de "carga" poderia envolver somente os segmentos COURSE, e uma atualização subsequente poderia ser rodada para inserir todos os segmentos dependentes como uma operação separada.

20.7 Daremos um esboço de uma possível abordagem (com agradecimentos a Don Chamberlin).



LDBs:

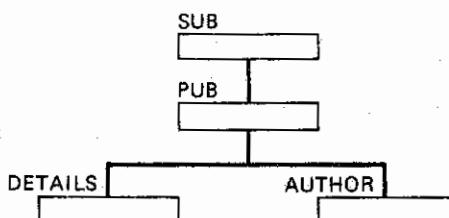


20.8 Para aplicações somente de recuperação, não há problema. O efeito sobre aplicações que executam operações de atualização sobre qualquer dos bancos de dados (PDBs ou LDBs) irá depender das regras de inserção/remoção/substituição especificadas para os segmentos envolvidos. Veja [19.1].

CAPÍTULO 21: ÍNDICES SECUNDÁRIOS DO IMS

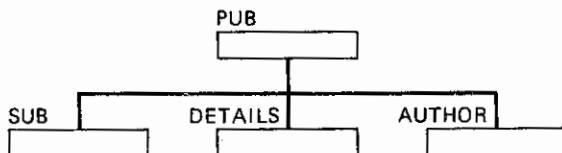
21.1 O usuário vê

(a)

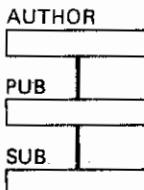


Neste caso, a estrutura hierárquica está inalterada.

(b)



(c)



Em todos os três casos (a), (b), e (c), a estrutura é vista na sequência AUTHNAME, com tantas ocorrências de raiz quantas forem as ocorrências de AUTHOR no banco de dados original e, para cada ocorrência de taiz, a quantidade correspondente de ocorrências dos dependentes. Veja a resposta ao Exercício 21.6.

21.2 DBD índice

```
DBD      NAME=AUTHXDBD, ACCESS=INDEX
SEG      NAME=XSEG, BYTES=16
FIELD    NAME=(AUTHNAME, SEQ, M), BYTES=16, START=1
LCHILD   NAME=(PUB, PUBDBD), INDEX=XAUTH
```

No PUBDBD são necessárias as duas instruções seguintes para o segmento PUB:

```
LCHILD NAME=(XSEG, AUTHXDBD), POINTER=INDX
XFLD   NAME=XAUTH, SRCH=AUTHNAME, SEGMENT=AUTHOR
```

PCB correspondente:

```
PCB      TYPE=DB, ..., KEYLEN=32, PROCSEQ=AUTHXDBD
SENSEG  NAME=PUB
SENSEG  NAME=SUB, PARENT=PUB
SENSEG  NAME=DETAILS, PARENT=PUB
SENSEG  NAME=AUTHOR, PARENT=PUB
```

21.3 GU PUB(XAUTH='ADAMS')

 TST if not found, exit

 print PUBNAME

 GN PUB(XAUTH='ADAMS')

 go to TST

21.4 PUB : AUTHNAME (i.e., XAUTH)

 SUB : AUTHNAME followed by SUB#

 DETAILS: AUTHNAME followed by DATE

 AUTHOR : AUTHNAME followed by AUTHNAME

 (the two names are not necessarily the same)

21.5 Get : no restrictions

 Insert : not allowed for PUB or SUB

 Delete : not allowed for PUB or SUB

 Replace: not allowed for PUBNAME, SUB#, DATE, AUTHNAME

21.6 PUB : 12000
 SUB : 12000
 DETAILS: 18000
 AUTHOR : 14400

21.7 Um possível método para suportar acesso seqüencial por chave a segmentos raiz em HDAM.

CAPÍTULO 24: ESTRUTURA DE DADOS DBTG

24.1 (Comparar com a resposta ao Exercício 16.2)

SCHEMA NAME IS PUB-SCHEMA

RECORD NAME IS SUB;
DUPLICATES ARE NOT ALLOWED FOR SUBNO IN SUB.
02 SUBNO ; TYPE IS CHARACTER 7.
02 SUBNAME ; TYPE IS CHARACTER 38.

RECORD NAME IS PUB;
 02 PUBNAME ; TYPE IS CHARACTER 44.
 02 AMFLAG ; TYPE IS BIT 1.

RECORD NAME IS DETAILS;
 02 DATE ; TYPE IS CHARACTER 6.
 02 PHJV ; TYPE IS CHARACTER 19.

RECORD NAME IS AUTHOR;
 02 AUTHNAME ; TYPE IS CHARACTER 16.
 02 AUTHADDR ; TYPE IS CHARACTER 34.

```
SET NAME IS SUBJECTS;
OWNER IS SYSTEM;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS SUB;
INSERTION IS AUTOMATIC
RETENTION IS FIXED;
KEY IS ASCENDING SURNO IN SUB.
```

```
SET NAME IS SUBPUB;
OWNER IS SUB;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE ALLOWED.
MEMBER IS PUB;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    KEY IS ASCENDING PUBNAME IN PUB;
    SET SELECTION IS BY APPLICATION. (default, not specified
                                explicitly)
```

```
SET NAME IS PUBDET;
OWNER IS PUB;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE ALLOWED.
MEMBER IS DETAILS;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    KEY IS DESCENDING DATE IN DETAILS;
SET SELECTION IS BY APPLICATION. (default, not specified
                                explicitly)
```

```
SET NAME IS PUBAUTH;
OWNER IS PUB;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS AUTHOR;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    KEY IS ASCENDING AUTHNAME IN AUTHOR;
    SET SELECTION IS BY APPLICATION. (default, not specified
                                explicitly)
```

Notas

1. AMFLAG foi declarado como BIT 1 somente para mostrar que BIT é um tipo de dado permissível.
2. A ordenação do conjunto PUBDET foi especificada como seqüência descendente de data simplesmente para mostrar que isto pode ser feito. Um dispositivo como esse em IMS teria simplificado consideravelmente o Exercício 18.7.
3. O conjunto singular SUBJECTS foi introduzido para que as ocorrências SUB possam ser ordenadas por SUBNO (como no banco de dados IMS correspondente).
4. Foi especificada para todos os casos a classe de membro AUTOMATIC FIXED, para fazer com que a estrutura DBTG reflita a estrutura IMS correspondente da forma mais próxima possível (onde nenhum segmento filho pode existir independentemente do seu pai).
5. DUPLICATES ARE ALLOWED significa que, por exemplo, duas publicações sobre o mesmo assunto podem ter o mesmo título, e que a posição relativa de dois registros membro com a mesma chave de controle de ordenação será determinada pelo DBMS, não pelo usuário. Para informações sobre outras opções DUPLICATES, veja [23.4].
6. SET SELECTION para os três conjuntos não singulares está mostrado como BY APPLICATION, para acompanhar de perto a situação em IMS. (BY APPLICATION está realmente especificado através da *omissão* da cláusula SET SELECTION; mas a mostramos explicitamente para maior clareza.)

24.2 SCHEMA NAME IS MANAGERIAL-STRUCTURE.

```
RECORD NAME IS EMP;
DUPLICATES ARE NOT ALLOWED FOR ENO IN EMP.
02 ENO . . .

RECORD NAME IS LINK.

SET NAME IS EL;
. . .
OWNER IS EMP;
MEMBER IS LINK;
. . .

SET NAME IS LE;
. . .
OWNER IS LINK;
MEMBER IS EMP;
. . .
```

Observe que a classe de membro tem que ser MANUAL em pelo menos um desses dois conjuntos (por quê?).

24.3 SCHEMA NAME IS S-P-J-SCHEMA.

RECORD NAME IS S;
DUPLICATES ARE NOT ALLOWED FOR SNO IN S.
02 SNO ; TYPE IS CHARACTER 5.
02 SNAME ; TYPE IS CHARACTER 20.
02 STATUS ; TYPE IS FIXED DECIMAL 3.
02 CITY ; TYPE IS CHARACTER 15.

RECORD NAME IS P;
DUPLICATES ARE NOT ALLOWED FOR PNO IN P.
02 PNO ; TYPE IS CHARACTER 6.
02 PNAME ; TYPE IS CHARACTER 20.
02 COLOR ; TYPE IS CHARACTER 6.
02 WEIGHT ; TYPE IS FIXED DECIMAL 4.

RECORD NAME IS J;
DUPLICATES ARE NOT ALLOWED FOR JNO IN J.
02 JNO ; TYPE IS CHARACTER 4.
02 JNAME ; TYPE IS CHARACTER 20.
02 CITY ; TYPE IS CHARACTER 15.

RECORD NAME IS SPJ;
DUPLICATES ARE NOT ALLOWED FOR SNO IN SPJ,
PNO IN SPJ,
JNO IN SPJ.
02 SNO ; TYPE IS CHARACTER 5.
02 PNO ; TYPE IS CHARACTER 6.
02 JNO ; TYPE IS CHARACTER 4.
02 QTY ; TYPE IS FIXED DECIMAL 5.

SET NAME IS S-SPJ;
OWNER IS S;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS SPJ;
INSERTION IS AUTOMATIC
RETENTION IS FIXED;
KEY IS ASCENDING PNO IN SPJ, JNO IN SPJ;
SET SELECTION IS BY VALUE OF SNO IN S.

SET NAME IS S-SET;
OWNER IS SYSTEM;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS S;
INSERTION IS AUTOMATIC
RETENTION IS FIXED;
KEY IS ASCENDING SNO IN S.

SET NAME IS P-SPJ;
OWNER IS P;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS SPJ;
INSERTION IS AUTOMATIC
RETENTION IS FIXED;
KEY IS ASCENDING JNO IN SPJ, SNO IN SPJ;
SET SELECTION IS BY VALUE OF PNO IN P.

```
SET NAME IS P-SET;
OWNER IS SYSTEM;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS P;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    KEY IS ASCENDING PNO IN P.
```

```
SET NAME IS J-SPJ;
OWNER IS J;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS SPJ;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    KEY IS ASCENDING SNO IN SPJ, PNO IN SPJ;
    SET SELECTION IS BY VALUE OF JNO IN J.
```

```
SET NAME IS J-SET;
OWNER IS SYSTEM;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS J;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    KEY IS ASCENDING JNO IN J.
```

24.4 SCHEMA NAME IS PARTS-AND-COMPONENTS.

```
RECORD NAME IS PART;
    DUPLICATES ARE NOT ALLOWED FOR PNO IN PART.
    02 PNO . . .
```

```
RECORD NAME IS LINK.
    02 QTY. . .
```

```
SET NAME IS BM;
    .
    .
    OWNER IS PART;
    MEMBER IS LINK;
    .
    .
```

```
SET NAME IS WU;
    .
    .
    OWNER IS PART;
    MEMBER IS LINK;
    .
    .
```

24.5 (a) SCHEMA NAME IS AREA-BIRD-SURVEY.

```
RECORD NAME IS AREA-REC;
    DUPLICATES ARE NOT ALLOWED FOR ANO IN AREA-REC.
    02 ANO ; TYPE IS CHARACTER 3.
    02 ANAME ; TYPE IS CHARACTER 24.
    02 ADESCN ; TYPE IS CHARACTER 473.
```

```
RECORD NAME IS BIRD-REC;
DUPLICATES ARE NOT ALLOWED FOR BNAME IN BIRD-REC;
DUPLICATES ARE NOT ALLOWED FOR SNAME IN BIRD-REC.
02 BNAME ; TYPE IS CHARACTER 44.
02 SNAME ; TYPE IS CHARACTER 44.
02 BDESCN ; TYPE IS CHARACTER 412.
```

```
RECORD NAME IS SIGHTING.
02 DATE ; TYPE IS CHARACTER 6.
02 REMARKS; TYPE IS CHARACTER 494.
```

```
SET NAME IS AREA-SET;
OWNER IS SYSTEM;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS AREA-REC;
INSERTION IS AUTOMATIC
RETENTION IS FIXED;
KEY IS ASCENDING ANO IN AREA-REC.
```

```
SET NAME IS BIRD-SET;
OWNER IS SYSTEM;
ORDER IS SORTED BY DEFINED KEYS
DUPLICATES ARE NOT ALLOWED.
MEMBER IS BIRD-REC;
INSERTION IS AUTOMATIC
RETENTION IS FIXED;
KEY IS ASCENDING SNAME IN BIRD-REC.
```

```
SET NAME IS AREA-SIGHTINGS;
OWNER IS AREA-REC;
ORDER IS NEXT.
MEMBER IS SIGHTING;
INSERTION IS AUTOMATIC
RETENTION IS MANDATORY;
SET SELECTION IS BY APPLICATION.
```

```
SET NAME IS BIRD-SIGHTINGS;
OWNER IS BIRD-REC;
ORDER IS NEXT.
MEMBER IS SIGHTING;
INSERTION IS AUTOMATIC
RETENTION IS MANDATORY;
SET SELECTION IS BY APPLICATION.
```

Uma vez que o registro SIGHTING não contém um item de dado correspondente a ANO IN AREA_REC ou um item de dado correspondente a SNAME (ou BNAME) em BIRD_REC, não é possível definir-se chaves de controle de ordenação para SIGHTINGS em relação aos conjuntos BIRD_SIGHTINGS e AREA_SIGHTINGS. Torna-se portanto responsabilidade do programador manter os registros SIGHTING na seqüência de nome de pássaro dentro da área, e seqüência de número de área dentro de pássaros, se forem necessárias essas ordenações. ORDER IS NEXT significa que o programador tem que selecionar, por meio de procedimentos, o precedente do novo registro no conjunto antes de criar aquele novo registro.

- (b) Devem ser feitas as seguintes alterações no esquema (a).
- Eliminar DATE do registro SIGHTING.

- Introduzir um novo tipo de registro:

```
RECORD NAME IS DATE-REC;
DUPLICATES ARE NOT ALLOWED FOR DATE IN DATE-REC.
02 DATE      ; TYPE IS CHARACTER 6.
```

- Introduzir um novo tipo de conjunto:

```
SET NAME IS DATE-SIGHTINGS;
OWNER IS DATE-REC;
ORDER IS NEXT.
MEMBER IS SIGHTING;
INSERTION IS AUTOMATIC
RETENTION IS MANDATORY;
SET SELECTION IS BY APPLICATION.
```

CAPÍTULO 25: O NÍVEL EXTERNO DO DBTG

25.1 Algumas observações úteis. Se O for antigo, qualquer programa que esteja correntemente executando operações de ERASE sobre O terá *provavelmente* que ser modificado (juntamente com o subschema correspondente). Se M for antigo e se sua classe de membro for AUTOMATIC, qualquer programa correntemente executando operações de STORE sobre M terá que ser modificado (juntamente com o subschema correspondente).

CAPÍTULO 26: MANIPULAÇÃO DE DADOS DBTG

A tabela de correntes da Seção 26.2 deve ser completada como a seguir.

Corrente da unidade de corrida	P 'P2'
Ocorrência S corrente	S 'S4'
Ocorrência P corrente	P 'P2'
Ocorrência SP corrente	SP 'S4/P2/200'
Registro corrente do conjunto S-SP	SP 'S4/P2/200' (membro)
Registro conjunto do conjunto P-SP	P 'P2' (dono)
Ocorrência S-SP corrente	do dono S 'S4'
Ocorrência P-SP corrente	do dono P 'P2'
Registro corrente do realm S-SP-P	P 'P2'

Nas respostas a seguir estaremos supondo que os procedimentos declarativos dos manuseios de exceções foram estabelecidos como no exemplo na Seção 26.3.

26.1.1 MOVE 'NO' TO NOTFOUND

```
MOVE 'J1' TO JNO IN J
FIND ANY J USING JNO IN J
IF NOTFOUND = 'NO'
  MOVE 'NO' TO EOF
  FIND FIRST SPJ WITHIN J-SPJ
  PERFORM UNTIL EOF = 'YES'
    GET SPJ
    (Add SNO IN SPJ to result list
     unless already present)
    FIND NEXT SPJ WITHIN J-SPJ
  END-PERFORM
END-IF
```

26.1.2 MOVE 'NO' TO EOF

```
MOVE 'J1' TO JNO IN J
MOVE 'P1' TO PNO IN SPJ
FIND SPJ WITHIN J-SPJ USING PNO IN SPJ
PERFORM UNTIL EOF = 'YES'
```

```
GET SPJ
  (add SNO IN SPJ to result list)
  FIND DUPLICATE WITHIN J-SPJ USING PNO IN SPJ
END-PERFORM
```

26.1.3 MOVE 'NO' TO NOTFOUND
MOVE 'J1' TO JNO IN J
FIND ANY J USING JNO IN J
IF NOTFOUND = 'NO'
 MOVE 'NO' TO EOF
 FIND FIRST SPJ WITHIN J-SPJ
 PERFORM UNTIL EOF = 'YES'
 GET SPJ
 FIND OWNER IN P-SPJ
 GET P
 IF COLOR IN P = 'RED'
 (add SNO IN SPJ to result list
 unless already present)
 END-IF
 END-PERFORM
END-IF

26.1.4 Primeiramente construímos uma tabela LONJNO contendo valores JNO dos projetos em Londres.

```
MOVE 'NO' TO EOF
MOVE 'LONDON' TO CITY IN J
FIND J WITHIN J-SET USING CITY IN J
PERFORM VARYING I FROM 1 BY 1 UNTIL EOF = 'YES'
  GET J
  MOVE JNO IN J TO LONJNO (I)
  FIND DUPLICATE WITHIN J-SET USING CITY IN J
END-PERFORM
```

Depois fazemos uma varredura nas ocorrências P, procurando peças para as quais existam ocorrências SPJ ligando a peça a cada um dos N projetos em Londres encontrados.

```
IF I > 1
  SUBTRACT 1 FROM I GIVING N
  MOVE 'NO' TO EOF
  FIND FIRST P WITHIN P-SET
  PERFORM UNTIL EOF = 'YES'
    GET P
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > N
      OR EOF = 'YES'
    MOVE LONJNO (I) TO JNO IN SPJ
    FIND SPJ WITHIN P-SPJ CURRENT USING JNO IN SPJ
  END-PERFORM
  IF EOF = 'YES'
    MOVE 'NO' TO EOF
    (add PNO IN P to result list)
  END-IF
  FIND NEXT P WITHIN P-SET
END-PERFORM
END-IF
```

26.1.5 MOVE 'NO' TO EOF

```

FIND FIRST J WITHIN J-SET
PERFORM UNTIL EOF = 'YES'
  GET J
  MOVE 'NO' TO EOF
  MOVE 'NO' TO BADJ
  FIND FIRST SPJ WITHIN J-SPJ
  PERFORM UNTIL EOF = 'YES' OR BADJ = 'YES'
    FIND OWNER WITHIN P-SPJ
    GET P
    IF COLOR IN P = 'RED'
      MOVE 'YES' TO BADJ
    ELSE
      FIND OWNER WITHIN S-SPJ
      GET S
      IF CITY IN S = 'LONDON'
        MOVE 'YES' TO BADJ
      ELSE
        FIND NEXT SPJ WITHIN J-SPJ
      END-IF
    END-IF
  END-PERFORM
  IF EOF = 'YES'
    MOVE 'NO' TO EOF
    (add JNO IN J to result list)
  END-IF
  FIND NEXT J WITHIN J-SET
END-PERFORM

```

26.1.6 Usamos uma lista de retenção, KX, de comprimento 2. Nossa solução poderia ser tornada mais eficiente se as duplicatas fossem eliminadas o mais cedo possível, ao invés de somente no estágio final.

```

LD KX LIMIT IS 2.

MOVE 'NO' TO NOTFOUND
MOVE 'S1' TO SNO IN S
FIND ANY S USING SNO IN S
IF NOTFOUND = 'NO'
  MOVE 'NO' TO EOF
  FIND FIRST SPJ WITHIN S-SPJ
  PERFORM UNTIL EOF = 'YES'
    KEEP USING KX
    FIND OWNER WITHIN P-SPJ
    FIND FIRST SPJ WITHIN P-SPJ
    PERFORM UNTIL EOF = 'YES'
      KEEP USING KX
      FIND OWNER WITHIN S-SPJ
      FIND FIRST SPJ WITHIN S-SPJ
      PERFORM UNTIL EOF = 'YES'
        GET SPJ
        (add PNO IN SPJ to result list
         unless already present)
        FIND NEXT SPJ WITHIN S-SPJ
  END-PERFORM
  MOVE 'NO' TO EOF
  FIND LAST WITHIN KX
  FREE LAST WITHIN KX
  FIND NEXT SPJ WITHIN P-SPJ

```

```
END-PERFORM  
MOVE 'NO' TO EOF  
FIND LAST WITHIN KX  
FREE LAST WITHIN KX  
FIND NEXT SPJ WITHIN S-SPJ  
END-PERFORM  
END-IF
```

26.1.7 MOVE 'NO' TO EOF
FIND FIRST S WITHIN S-SET
PERFORM UNTIL EOF = 'YES'
 GET S
 FIND FIRST SPJ WITHIN S-SPJ
 PERFORM UNTIL EOF = 'YES'
 FIND OWNER IN J-SPJ
 GET J
 (add <CITY IN S,CITY IN J> to result list
 unless already present)
 FIND NEXT SPJ WITHIN S-SPJ
 END-PERFORM
MOVE 'NO' TO EOF
FIND NEXT S WITHIN S-SET
END-PERFORM

26.1.8 Este problema é surpreendentemente difícil de ser eficientemente manuseado. Uma solução recupera todo o conjunto de ocorrências P uma de cada vez, não sendo um procedimento muito deseável. Mas o Formato 2 de FIND não pode ser usado (por quê?), e o Formato 6 de FIND seria ainda menos eficiente (por quê?). (*Sugestão:* Para ver por que o Formato 2 de FIND não pode ser usado, considere o que aconteceria se as peças vermelhas tivessem que ser removidas, não apenas modificadas.)

```
MOVE 'NO' TO EOF  
FIND FIRST P WITHIN P-SET  
PERFORM UNTIL EOF = 'YES'  
    GET COLOR IN P  
    IF COLOR IN P = 'RED'  
        MOVE 'ORANGE' TO COLOR IN P  
        MODIFY COLOR IN P  
    END-IF  
    FIND NEXT P WITHIN P-SET  
END-PERFORM  
COMMIT
```

26.1.9 Estamos supondo que existe uma ocorrência SPJ para S1/P1/J1; entretanto, não supomos que uma ocorrência SPJ para S2/P1/J1 já não exista. Portanto, a primeira etapa é dar um FIND e um GET na ocorrência S2/P1/J1 se ela existir, para que a QTY apropriada possa ser adicionada ao valor QTY de S1/P1/J1, formando o novo valor de QTY de S2/P1/J1. Se não existir, naturalmente, nada terá que ser adicionado ao valor de QTY de S1/P1/J1. Também removemos a ocorrência SPJ para S2/P1/J1 se ela existir.

```
MOVE 'NO' TO EOF  
MOVE 'S2' TO SNO IN S  
MOVE 'P1' TO PNO IN SPJ  
MOVE 'J1' TO JNO IN SPJ  
FIND SPJ WITHIN S-SPJ USING PNO IN SPJ, JNO IN SPJ  
IF EOF = 'NO'  
    GET QTY IN SPJ  
    MOVE QTY IN SPJ TO TEMPQTY  
    ERASE SPJ
```

```
ELSE
  MOVE 'NO' TO EOF
  MOVE ZERO TO TEMPQTY
END-IF
```

Possseguimos agora emitindo FIND, MODIFY e RECONNECT para a ocorrência S1/P1/J1 de SPJ.

```
MOVE 'S1' TO SNO IN S
FIND SPJ WITHIN S-SPJ USING PNO IN SPJ, JNO IN SPJ
GET QTY IN SPJ
ADD TEMPQTY TO QTY IN SPJ
MOVE 'S2' TO SNO IN SPJ
MODIFY SNO IN SPJ, QTY IN SPJ
MOVE 'S2' TO SNO IN S
RECONNECT SPJ WITHIN S-SPJ
COMMIT
```

26.3 A classe de membro de LINK em EL deveria ser provavelmente AUTOMATIC, pois não seria razoável que uma ocorrência LINK não estivesse conectada a uma ocorrência EMP superior (gerente). Supondo que a classe de membro de LINK seja AUTOMATIC, então a classe de membro de EMP em LE tem que ser MANUAL, para permitir que pelo menos uma ocorrência de EMP (no topo de árvore) não esteja subordinada a nenhuma outra.

Estamos supondo que a SET SELECTION de EL seja BY VALUE OF ENO IN EMP, e que a SET SELECTION de LE seja BY APPLICATION.

```
MOVE 'E8' TO ENO IN EMP
STORE LINK (no UWA data for this record)
MOVE 'E15' TO ENO IN EMP
MOVE . . . (create E15 EMP occurrence in UWA)
STORE EMP
CONNECT EMP TO LE
COMMIT
```

26.4.1 MOVE 'NO' TO EOF
FIND FIRST PART WITHIN PART-SET
PERFORM UNTIL EOF = 'YES'
 GET PART
 (print PNO IN PART)
 FIND FIRST LINK WITHIN BM
 PERFORM UNTIL EOF = 'YES'
 FIND OWNER WITHIN WU
 RETAINING PART-SET, BM CURRENCY
 GET PART
 (print PNO IN PART, i.e., component no.)
 FIND NEXT LINK WITHIN BM
END-PERFORM
MOVE 'NO' TO EOF
FIND CURRENT PART WITHIN PART-SET
FIND FIRST LINK WITHIN WU
PERFORM UNTIL EOF = 'YES'
 FIND OWNER WITHIN BM
 RETAINING PART-SET, WU CURRENCY
 GET PART
 (print PNO IN PART, i.e., assembly no.)
 FIND NEXT LINK WITHIN WU
END-PERFORM
MOVE 'NO' TO EOF
FIND NEXT PART WITHIN PART-SET
END-PERFORM

26.4.2 LD STACK LIMIT IS 10. (O limite de 10 é arbitrário)

```
MOVE 'NO' TO NOTFOUND
MOVE GIVEN-PNO TO PNO IN PART
FIND ANY PART USING PNO IN PART
IF NOTFOUND = 'NO'
    GET PART
    (print PNO IN PART)
    MOVE ZERO TO STACK-DEPTH
    MOVE 'NO' TO EOF
    PERFORM UNTIL STACK-DEPTH < ZERO
        FIND NEXT LINK WITHIN BM
        IF EOF = 'NO'
            GET LINK
            KEEP USING STACK
            ADD 1 TO STACK-DEPTH
            FIND OWNER WITHIN WU
            GET PART
            (print PNO IN PART)
        ELSE
            IF STACK-DEPTH > ZERO
                FIND LAST WITHIN STACK
                FREE LAST WITHIN STACK
                MOVE 'NO' TO EOF
            END-IF
            SUBTRACT 1 FROM STACK-DEPTH
        END-IF
    END-PERFORM
END-IF
```

CAPÍTULO 27: A LINGUAGEM UNIFICADA DE BANCOS DE DADOS

Apresentaremos as versões COBOL do procedimento de matrícula (de acordo com as Figs. 27.5, 27.6, e 27.7; veja também as contabilizações comparativas na Fig. 27.9).

Relacional:

```
MOVE 'YES' TO APPLICATION-OK
PERFORM PREREQ MATCHING GIVEN
    UNTIL APPLICATION-OK = 'NO'
    IF EXISTS (STUDENT MATCHING GIVEN ON EMPNO
                WHERE COURSENO OF STUDENT = PRENO OF PREREQ)
        CONTINUE
    ELSE MOVE 'NO' TO APPLICATION-OK
    END-IF
END-PERFORM
IF APPLICATION-OK = 'YES'
    CREATE STUDENT FROM GIVEN, SPACES
END-IF
```

Hierárquica:

```
MOVE 'YES' TO APPLICATION-OK
FIND UNIQUE (COURSE MATCHING GIVEN)
PERFORM PREREQ UNDER COURSE
    UNTIL APPLICATION-OK = 'NO'
    IF EXISTS (STUDENT MATCHING GIVEN
                WHERE UNIQUE (COURSENO OF COURSE OVER STUDENT)
                = PRENO OF PREREQ)
```

```

CONTINUE
ELSE MOVE 'NO' TO APPLICATION-OK
END-IF
END-PERFORM
IF APPLICATION-OK = 'YES'
  FIND UNIQUE (OFFERING MATCHING GIVEN UNDER COURSE)
  CREATE STUDENT FROM EMPNO OF GIVEN, SPACES
    CONNECTING UNDER OFFERING
END-IF

```

Em rede:

```

MOVE 'YES' TO APPLICATION-OK
FIND UNIQUE (COURSE MATCHING GIVEN)
PERFORM PREREQ UNDER COURSE VIA HASPRE
  UNTIL APPLICATION-OK = 'NO'
  IF EXISTS (STUDENT WHERE
    UNIQUE (EMPNO OF EMPLOYEE OVER STUDENT)
      = EMPNO OF GIVEN
    AND UNIQUE (COURSENO OF COURSE OVER STUDENT)
      =
    UNIQUE (COURSENO OF COURSE OVER PREREQ
      VIA PREOF))

  CONTINUE
  ELSE MOVE 'NO' TO APPLICATION-OK
END-IF
END-PERFORM
IF APPLICATION-OK = 'YES'
  FIND UNIQUE (OFFERING MATCHING GIVEN UNDER COURSE)
  FIND UNIQUE (EMPLOYEE MATCHING GIVEN)
  CREATE STUDENT FROM EMPNO OF GIVEN, SPACES
    CONNECTING UNDER OFFERING,
    UNDER EMPLOYEE
END-IF

```

Nas soluções a seguir, usaremos "print" como uma abreviação genérica dos operadores PUT SKIP LIST e outros do PL/I.

27.1.1 Mostraremos várias soluções deste exercício, para ilustrar diversos dispositivos UDL.

Relacional, nível de registro, primeira solução:

```

DO SPJ WHERE SPJ.J# = 'J1';
  print SPJ.S#;
END;

```

Se quisermos garantir que os resultados sejam impressos, digamos, em ordem ascendente de número de fornecedor, podemos incluir a especificação ORDER (UP SPJ.S#) na instrução DO. Na ausência de uma especificação ORDER, a interação é executada de acordo com a ordenação declarada do conjunto base de apoio (SPJSET) – isto é, ordenação definida pelo sistema (*default*), neste caso particular. Ignoraremos ORDER na maioria das nossas soluções.

Relacional, nível de registro, segunda solução

```

DO SPJ.S# WHERE SPJ.J# = 'J1'
  NOSET ASSIGNTO(TEMP#);
  print TEMP#;
END;

```

Aqui a interação é executada sobre um “conjunto seccionado” – isto é, sobre um conjunto de campos S#, ao invés de sobre um conjunto de registros SPJ. “SPJ.S# WHERE . . .” é uma expressão de conjunto (na realidade uma “referência de conjunto seccionado”) que não designa um conjunto de registros existentes no banco de dados, mas sim um conjunto de registros que são derivados de alguma forma desses registros. Não é permitida a colocação de um cursor indicando a localização de um registro derivado; daí a especificação NOSET e também a especificação ASSIGNTO, que é requerida e que faz com que uma cópia do registro corrente seja designada à variável indicada.

Relacional, nível de registro, terceira solução:

Esta solução elimina duplicatas de números de fornecedores.

```
DO DISTINCT(SPJ.S# WHERE SPJ.J# = 'J1')
    NOSET ASSIGNTO(TEMP#);
    print TEMP#;
END;
```

NOSET e ASSIGNTO são requeridos sempre que o escopo do loop DO for especificado como uma referência DISTINCT.

Relacional, nível de conjunto:

```
print DISTINCT(SPJ.S# WHERE SPJ.J# = 'J1');
```

Rede, nível de registro, primeira solução:

```
FIND UNIQUE(J WHERE J.J# = 'J1');
DO SPJ UNDER J;
    print SPJ.S#;
END;
```

Rede, nível de registro, segunda solução:

```
DO SPJ WHERE UNIQUE(J.J# OVER SPJ) = 'J1';
    print UNIQUE(S.S# OVER SPJ);
END;
```

Esta solução é provavelmente menos eficiente do que a anterior, embora a implementação só necessite pesquisar os registros SPJ que estão “sob o projeto J1”.

Rede, nível de registro, terceira solução:

```
DO S WHERE EXISTS(SPJ UNDER S
                  WHERE UNIQUE(J.J# OVER SPJ) = 'J1');
    print S.S#;
END;
```

Rede, nível de conjunto:

```
print S.S# WHERE EXISTS(SPJ UNDER S
                  WHERE UNIQUE(J.J# OVER SPJ) = 'J1');
```

Para as questões restantes, daremos somente soluções a nível de registro.

27.1.2

Relacional:

```
O SPJ WHERE SPJ.J# = 'J1' & SPJ.P# = 'P1';
    print SPJ.S#;
END;
```

Em rede:

```
DO SPJ WHERE UNIQUE(J.J# OVER SPJ) = 'J1'
    & UNIQUE(P.P# OVER SPJ) = 'P1';
    print UNIQUE(S.S# OVER SPJ);
END;
```

27.1.3

Relacional:

```
DO S.S# WHERE EXISTS(SPJ MATCHING S
                     WHERE SPJ.J# = 'J1' &
                       UNIQUE(P.COLOR MATCHING SPJ) = 'RED')
NOSET ASSIGNTO(TEMP#);
print TEMP#;
END;
```

Em rede:

```
DO S.S# WHERE EXISTS(SPJ UNDER S
                     WHERE UNIQUE(J.J# OVER SPJ) = 'J1' &
                       UNIQUE(P.COLOR OVER SPJ) = 'RED')
NOSET ASSIGNTO(TEMP#);
print TEMP#;
END;
```

27.1.4

Relacional:

```
DO P WHERE NOT EXISTS(J WHERE J.CITY = 'LONDON' &
                      NOT EXISTS(SPJ WHERE SPJ.P# = P.P# &
                                 SPJ.J# = J.J#));
print P.P#;
END;
```

Em rede:

Pode ser obtida uma solução em rede a partir da solução relacional, substituindo-se SPJ.P# e SPJ.J# por UNIQUE (P.P# OVER SPJ) e UNIQUE (J.J# OVER SPJ), respectivamente.

27.1.5

Relacional:

```
DO J WHERE NOT EXISTS(SPJ MATCHING J
                     WHERE UNIQUE(P.COLOR MATCHING SPJ) = 'RED'
                       & UNIQUE(S.CITY MATCHING SPJ) = 'LONDON');
print J.J#;
END;
```

Em rede:

Pode ser obtida uma solução em rede a partir da solução relacional substituindo-se o primeiro MATCHING por UNDER e os outros dois MATCHINGS por OVER.

27.1.6

Sejam X, Y, Z os cursores do tipo de registro SPJ.

Relacional:

```
DO P WHERE EXISTS(X->SPJ MATCHING P WHERE
                     EXISTS(Y->SPJ MATCHING X->SPJ ON S# WHERE
                           EXISTS(Z->SPJ MATCHING Y->SPJ ON P#
                                 WHERE Z->SPJ.S# = 'S1'))));
print P.P#;
END;
```

Em rede:

```
DO P WHERE EXISTS(X→SPJ UNDER P WHERE
    EXISTS(Y→SPJ UNDER UNIQUE(S OVER X→SPJ) WHERE
        EXISTS(Z→SPJ UNDER UNIQUE(P OVER Y→SPJ)
            WHERE UNIQUE(S.S# OVER Z→SPJ)
                = 'S1')))

    print P.P#;
END;
```

27.1.7

Seja PAIR declarado como a seguir:

```
DCL 1 PAIR,
    2 SC . . . ;
    2 JC . . . ;
```

Relacional:

```
DO DISTINCT((S.CITY,J.CITY)
    WHERE EXISTS(SPJ WHERE SPJ.S# = S.S# &
        SPJ.J# = J.J#))
    NOSET ASSIGNTO(PAIR);

    print PAIR;
END;
```

Em rede:

Pode ser obtida uma solução em rede a partir da solução relacional substituindo-se SPJ.S# e SPJ.J# por UNIQUE (S.S# OVER SPJ) e UNIQUE (J.J# OVER SPJ), respectivamente.

27.1.8

Relacional/em rede, nível de registro:

```
DO P WHERE P.COLOR = 'RED';
    ASSIGN 'ORANGE' TO P.COLOR;
END;
COMMIT;
```

Relacional/em rede, nível de conjunto:

```
ASSIGN ('ORANGE' TO P.COLOR
    DO P WHERE P.COLOR = 'RED');
COMMIT;
```

A especificação DO embutida dentro do ASSIGN aqui é análoga a uma especificação DO embutida dentro de uma lista de dados nas instruções correntes GET e PUT do PL/I.

27.1.9

Relacional:

```
FIND UNIQUE(SPJ WHERE SPJ.S# = 'S1' &
    SPJ.P# = 'P1' &
    SPJ.J# = 'J1')
    SET (X);

FIND UNIQUE(SPJ WHERE SPJ.S# = 'S2' &
    SPJ.P# = 'P1' &
    SPJ.J# = 'J1')
    SET (Y)
```

```

FOUND
  DO;
    ASSIGN (X→SPJ.QTY+Y→SPJ.QTY) TO Y→SPJ.QTY;
    FREE X→SPJ;
  END;
NOTFOUND
  ASSIGN 'S2' TO X→SPJ.S#;
  COMMIT;

```

Em rede:

```

FIND UNIQUE(SPJ WHERE UNIQUE(S.S# OVER SPJ) = 'S1' &
            UNIQUE(P.P# OVER SPJ) = 'P1' &
            UNIQUE(J.J# OVER SPJ) = 'J1')
            SET (X);
FIND UNIQUE(SPJ WHERE UNIQUE(S.S# OVER SPJ) = 'S2' &
            UNIQUE(P.P# OVER SPJ) = 'P1' &
            UNIQUE(J.J# OVER SPJ) = 'J1')
            SET (Y)
FOUND
  DO;
    ASSIGN (X→SPJ.QTY+Y→SPJ.QTY) TO Y→SPJ.QTY;
    FREE X→SPJ;
  END;
NOTFOUND
  RECONNECT X→SPJ UNDER UNIQUE(S WHERE S.S# = 'S2');
  COMMIT;

```

27.2.1

Relacional:

```

DO PART IN PART_SET; /* iterate over all parts */
  print PART.P#;
  DO COMPONENT WHERE MAJORP# = PART.P#;
    print MINORP#;
  END;
  DO COMPONENT WHERE MINORP# = PART.P#;
    print MAJORP#;
  END;
END;

```

Em rede:

```

DO PART IN PART_SET;
  print PART.P#;
  DO COMPONENT UNDER PART VIA BM;
    print UNIQUE(PART.P# OVER COMPONENT VIA WU);
  END;
  DO COMPONENT UNDER PART VIA WU;
    print UNIQUE(PART.P# OVER COMPONENT VIA BM);
  END;
END;

```

27.2.2

Relacional:

```

CALL EXPLODE(GIVENP#);

EXPLODE:

```

```

PROC(UPPERP#) RECURSIVE;
DCL UPPERP# . . . ;
DCL X CURSOR RECTYPE(COMPONENT);
print UPPERP#;
DO COMPONENT WHERE MAJORP# = UPPERP# SET(X);
    CALL EXPLODE(X-MINORP#);
END;
END /* EXPLODE */;

```

Em rede:

O procedimento a seguir usa valores de cursor para comunicação entre níveis de explosão (diferentemente da solução relacional, que usa números de peças).

```

FIND UNIQUE(PART WHERE PART.P# = GIVENP#) SET(P);
CALL EXPLODE(P);

```

EXPLODE:

```

PROC(PU) RECURSIVE;
DCL (PU,PL) CURSOR RECTYPE(PART),
    X CURSOR RECTYPE(COMPONENT);
print PU-PART.P#;
DO COMPONENT UNDER PU-PART VIA BM SET(X);
    FIND UNIQUE(PART OVER X-COMPONENT VIA WU) SET(PL);
    CALL EXPLODE(PL);
END;
END /* EXPLODE */;

```

27.3

Hierárquico:

```

FIND UNIQUE(COURSE WHERE COURSE.COURSE# = GIVEN.COURSE#) SET(C1);
DO PREREQ UNDER C1-COURSE;
    FIND UNIQUE(COURSE WHERE COURSE.COURSE# = PREREQ.PRE#) SET(C2);
    DO OFFERING UNDER C2-COURSE;
        print OFFERING;
    END;
END;

```

27.4

(a) *Relacional:* A abordagem óbvia é mapear cada registro diretamente a um registro armazenado isomórfico, e adicionar um índice na chave primária de cada tipo de registro. De fato, como PREREQS e TEACHERs são “todo chave”, o índice sozinho seria suficiente nesses dois casos – não são necessários registros de dados; entretanto, vamos supor que existam registros de dados em todos os casos.

Hierárquico: Mapear cada conjunto agregado a um conjunto agregado armazenado. Cada registro pai tem um indicador de localização do primeiro filho. Cada registro filho tem um indicador da localização do próximo filho e um indicador da localização do pai. Também, indexar COURSES e EMPLOYEES em suas chaves primárias.

Em rede: Mesma abordagem do caso hierárquico.

(b) Seja:

- c = a quantidade de COURSES,
- e = a quantidade de EMPLOYEES,
- p = a quantidade média de PREREQs por COURSE,
- x = a quantidade média de OFFERINGS por COURSE,
- y = a quantidade média de STUDENTS por OFFERING, e
- z = a quantidade média de TEACHERs por OFFERING.

(Observe que a quantidade média de STUDENTS por EMPLOYEE será então cxz/e .)

Vamos supor indicadores de localização de 4 bytes; vamos supor também, por simplicidade, que um índice consiste somente de um conjunto de entradas, uma para cada registro indexado, e que cada entrada consiste exatamente de um valor de chave e um indicador de localização (isto é, ignoraremos níveis de índices acima do ‘conjunto índice’ [veja o Capítulo 2], e também o fato de que os índices são tipicamente comprimidos). Assim, os requisitos de espaço de memória são como a seguir.

Relacional:

COURSES	$c(36 + (3 + 4))$	= 43c
PREREQs	$cp(6 + (6 + 4))$	= 16cp
OFFERINGS	$cx(24 + (6 + 4))$	= 34cx
STUDENTS	$cxy(13 + (9 + 4))$	= 26cxy
TEACHERS	$cxz(12 + (12 + 4))$	= 28cxz
EMPLOYEES	$e(24 + (6 + 4))$	= 34e

$$\text{Total} = c(2x(13y + 14z + 17) + 16p + 43) + 34e \text{ bytes.}$$

(Se supusermos que PREREQs e TEACHERs são subsomados pelos índices correspondentes, este total se reduz de $6c(p + 2xz)$ bytes).

Hierárquico:

COURSES	$c(36 + 4 + 4 + (3 + 4))$	= 51c
PREREQs	$cp(3 + 4 + 4)$	= 11cp
OFFERINGS	$cx(21 + 4 + 4 + 4 + 4)$	= 37cx
STUDENTS	$cxy(7 + 4 + 4)$	= 15cxy
TEACHERS	$cxz(6 + 4 + 4)$	= 14cxz
EMPLOYEES	$e(24 + (6 + 4))$	= 34e

$$\text{Total} = c(x(15y + 14z + 37) + 11p + 51) + 34e \text{ bytes.}$$

Em rede:

COURSES	$c(36 + 4 + 4 + 4 + (3 + 4))$	= 55c
PREREQs	$cp(4 + 4 + 4 + 4)$	= 16cp
OFFERINGS	$cx(21 + 4 + 4 + 4 + 4 + 4)$	= 37cx
STUDENTS	$cxy(1 + 4 + 4 + 4 + 4 + 4)$	= 17cxy
TEACHERS	$cxz(4 + 4 + 4 + 4)$	= 16cxz
EMPLOYEES	$e(24 + 4 + 4 + (6 + 4))$	= 42e

$$\text{Total} = c(x(17y + 16z + 37) + 16p + 55) + 42e \text{ bytes.}$$

Como uma ilustração, suponhamos $c = 100$, $e = 1000$, $p = 3$, $x = 8$, $y = 16$, e $z = 1,5$. Então, as três expressões serão avaliadas como a seguir.

Relacional: 436,700 bytes

Hierárquico: 280,800 bytes

Em rede: 318,700 bytes

(c) Vamos nos concentrar na porção de *loop* do procedimento, que é executado p vezes em média (supondo-se que os empregados dados tenham participado de todos os cursos pré-requisito relevantes). Nos três casos, cada interação do *loop* envolve duas chamadas do DBMS; uma para mover para o próximo PREREQ e outra para verificar a existência de um STUDENT. Portanto, o número total de chamadas é $2p$ em qualquer caso. Entretanto, as chamadas em si vão se tornando progressivamente mais complexas, e o DBMS tem que executar correspondentemente mais trabalho, à medida que passamos de relações para hierarquias e para redes. As expressões derivadas abaixo dão uma indicação desse aumento de complexidade. Chamamos a atenção do leitor, no entanto, para que não tome esses números como medidas absolutas de desempenho.

Relacional: Suporemos (o que é razoável) que todas as entradas de índices para os PREREQS de um dado COURSE estejam em um mesmo bloco de índice. Então, cada entrada de PREREQ após a primeira envolverá exatamente um acesso direto ao banco de dados. (Uma implementação inteligente poderia perceber que o acesso ao registro de dados é desnecessário, pois os valores de dado já estão disponíveis na entrada do índice; mas não vamos supor esta implementação.) Passando agora ao teste de EXISTS, observamos que o teste é de um STUDENT tendo um determinado valor de chave primária, e que portanto o teste pode ser determinado somente pelo índice – não é necessário acesso ao dado em si. Além disso, é também provável que todas as entradas de índices de STUDENT para o COURSE dado estejam no mesmo bloco físico, mas não faremos esta hipótese. Portanto, nossa expressão final da quantidade de registros do banco de dados (registros de dados e registros de índices) que recebem acesso no loop é simplesmente $2p$ (sendo p para os PREREQS e p para os índices de registros STUDENT).

Hierárquico: Uma vez que cada PREREQ sob o COURSE dado indica diretamente a localização do próximo, podemos novamente supor que cada PREREQ envolve somente um acesso. O teste de EXISTS é, entretanto, mais complicado. Se supusermos que a implementação irá pesquisar a partir do topo da hierarquia, teremos:

- um acesso ao índice de COURSE na base de PREREQ.PRE #
- um acesso ao COURSE correspondente
- varredura de metade das x OFFERINGS correspondentes (em média)
- para cada –
 - um acesso a OFFERING
 - varredura de todos os y STUDENTS correspondentes
 - para cada –
 - um acesso a STUDENT

portanto, a expressão final é

$$p(1 + 1 + 1 + (x/2)(1 + y)) = ((xy/2) + (x/2) + 3)p.$$

Em rede: Novamente suporemos um acesso por PREREQ. Entretanto, existem agora duas estratégias para o teste de EXISTS, uma varrendo para baixo a partir do COURSE pré-requisito, e outra varrendo para baixo a partir do EMPLOYEE dado. Para a primeira, temos:

- Um acesso a COURSE via o indicador de localização pai de PREOF
- varredura de metade das x OFFERINGS correspondentes (em média)
- para cada –
 - um acesso a OFFERING
 - varredura de todos os y STUDENTS correspondentes
 - para cada –
 - um acesso a STUDENT
 - um acesso a EMPLOYEE via o indicador de localização pai de ATTENDS

Portanto, a primeira possibilidade é

$$p(1 + 1 + (x/2)(1 + y(1 + 1))) = (xy + (x/2) + 2)p.$$

Para a segunda estratégia, temos:

- varredura de metade dos cxy/e STUDENTS para o EMPLOYEE dado
- para cada –
 - um acesso a STUDENT
 - um acesso a OFFERING via o indicador de localização pai de HASSTU
 - um acesso a COURSE via o indicador de localização pai de HASOFFER

Portanto, a segunda possibilidade é

$$p(1 + (cxy/2e)(1 + 1 + 1)) = ((3cxy/2e) + 1)p.$$

Observe que as duas expressões do caso em rede seriam muito piores se não houvesse os indicadores de localização país.

Tomando os mesmos valores para c , e , p , x , e y como na parte (b) da resposta, essas expressões seriam avaliadas como se segue:

Relacional:	6
Hierárquico:	213
Em rede:	(1) 402
	(2) 61 no inteiro mais próximo.

Lista de Abreviaturas

Relacionamos abaixo algumas das abreviaturas mais importantes introduzidas no texto, com o seu significado.

ANSI/SPARC	literalmente, American National Standards Institute/Systems Planning and Requirements Committee; usada nas referências à arquitetura de banco de dados de três níveis descrita no Capítulo 1
BCNF	Forma normal de Boyce/Codd
DBA	administrador do banco de dados
DBD	descrição do banco de dados (IMS)
DBMS	database management system – sistema de gerência do banco de dados
DBTG	Data Base Task Group
DDL	linguagem de descrição de dados; linguagem de definição de dados
DEDB	banco de dados de entrada de dados (IMS)
DK/NF	forma normal domínio/chave
DML	linguagem de manipulação de dados
DSDL	linguagem de descrição de dados armazenados (DBTG)
DSG	grupos de arquivos (IMS)
DSL	sublinguagem de dados
FD	dependência funcional
FSA	argumento de pesquisa de campo (IMS)
GN	get next (IMS)
GNP	get next within parent (IMS)
GSAM	generalized sequential access method (IMS)
GU	get unique (IMS)
HDAM	método de acesso hierárquico direto (IMS)
HIDAM	método de acesso direto hierárquico indexado (IMS)
HISAM	método de acesso hierárquico seqüencial indexado (IMS)
HSAM	método de acesso hierárquico seqüencial
IMS	Information Management System

ISAM	método de acesso seqüencial indexado
JD	dependência de junção
LDB	banco de dados lógico (IMS)
LDBR	registro de banco de dados lógico (IMS)
MSDB	banco de dados na memória principal (IMS)
MVD	dependência de múltiplos valores
OSAM	método de acesso seqüencial de excedentes (IMS)
PCB	bloco de comunicação do programa (IMS)
PDB	banco de dados físico (IMS)
PDBR	registro de banco de dados físico (IMS)
PJ/NF	forma normal projeção/junção
PSB	bloco de especificação do programa (IMS)
QBE	Query By Example
QUEL	Query Language (INGRES)
RDS	sistema de dados relacional (Sistema R)
r-s-e	expressão de seleção de registro (DBTG)
RSI	interface de pesquisa à memória (Sistema R)
RSS	sistema de pesquisa à memória (Sistema R)
SHISAM	HISAM simples (IMS)
SHSAM	HSAM simples (IMS)
SQL	Structured Query Language (Sistema R)
SRA	endereço do registro armazenado
SSA	argumento de pesquisa de segmento (IMS)
TID	ID de tupla (Sistema R, INGRES)
UDL	linguagem unificada de bancos de dados
UFI	interface voltado para o usuário (Sistema R)
UWA	área de trabalho do usuário (DBTG)
VSAM	método de acesso à memória virtual
WFF	fórmula bem-formada
XD	indexado (IMS)
1NF	primeira forma normal
2NF	segunda forma normal
3NF	terceira forma normal
4NF	quarta forma normal
5NF	quinta forma normal (mesmo que PJ/NF)

Índice Analítico

*As entradas marcadas com um asterisco (por exemplo, *ADABAS) são sistemas de gerenciamento de bancos de dados ou sistemas de software de algum campo estreitamente relacionado.*

- Abordagem infológica, 50
- *ADABAS, 49
- *ADAM, 48
- Administrador do banco de dados, 29, 32, 45
 - responsabilidades, 45-46
- Administrador de dados, 354
- Agregado, 405
- Aho, A. V., 206, 242
- *Álgebra da informação, 207
- Álgebra relacional, 193, 196
 - operações primitivas, 203
- Algoritmo de redução de Codd, 204
- All (QBE), 174
- Allman, E., 219
- ALPHA (sublinguagem de dados), 89, 209
- Altman, E. B., 73
- American National Standards Institute, 49, 50
- AND dependente (IMS), 336
- AND independente (IMS), 336
- Anomalia, *veja* Anomalia de atualização
- Anomalia de atualização
 - hierarquia, 80-81
 - 1NF, 227
 - 2NF, 227
 - 3NF, 228
- ANSI, 49, 50
- ANSI/SPARC, 38
 - arquitetura, 38-46
- ANSI/X3, 49
- ANSI/X3/SPARC, *veja* ANSI/SPARC
- Antonacci, F., 142
- APL (A Programming Language), 142, 173
- *APL (Associative Programming Language), 354
- *AQL, 142
- Área (DEDB), 348
- Área de excedentes (HDAM), 293
- Área de recebimento da chave (IMS), 268
- Área de registro (DBTG), 354
- Área de trabalho do usuário (DBTG), 354, 379
- Área endereçável do segmento raiz (HDAM), 293
- Argumento de pesquisa de campo (IMS), 345
- Argumento de pesquisa de segmento (MS), 272-278
- Armadilha de conexão, 31
- Armstrong, W. W., 249
- Arnold, R. S., 142
- Arquivo (OS/VS), 286
- Arquivo armazenado, 35
 - RSS, 107-108, 165, 167
- *Arquivo de dados relacional, 207
- Arquivo diferencial, 74
- Árvore-B, 64-66, 71
- ASC/DESC (SQL), *veja* Ordenação
- ASCENDING/DESCENDING KEY (DBTG), *veja* Ordenação
- Ash, W., 207
- ASL, 154
- ASSIGN (UDL), 420
- ASSIGNTO (UDL), 493
- Associação, 30, 31, 89
- Astrahan, M. M., 73, 110, 111, 154
- Atualização (QBE), 184
- Atualização (UDL), 420
- Atualização de visão, 158, 163

- Atributo (de uma relação), 78, 95, 101
Atributo composto, 225
Atributo não-chave, 230
AUTOMATIC – classe de inserção (DBTG), 373
Autorização, 33
AVG (QBE), 182
AVG (SQL), 132
Axiomas (de dependências), 249, 250
Axiomas de Armstrong, 249
- Bachman, C. W., 91, 358, 376, 406, 436, 437
Banco de dados, 27, 29
compartilhado, 27, 32
em rede, 405
hierárquico, 405
integrado, 27
relacional, 100, 405
UDL, 404
visão do usuário, 24
Banco de dados armazenado, 44
Banco de dados compartilhado, *veja* Banco de dados
Banco de dados de entrada de dados, 343
Banco de dados de índice compartilhado (IMS), 339
Banco de dados distribuído, 47
Banco de dados em rede, 405
Banco de dados físico (IMS), 254, 257
Banco de dados hierárquico, 405
Banco de dados índice (IMS) HIDAM, 296-297, 302
indexação secundária, 332
Banco de dados lógico (IMS), 256, 265, 309
Banco de dados na memória (IMS), 343
Banco de dados relacional, 100
UDL, 404
Bandurski, A. E., 438
Bayer, R., 64, 72
BCNF, 233
Beitz, E. H., 208
Beeri, C., 206, 242, 249, 251
BEGIN TRANSACTION (SQL embutida), 150
Bernstein, P. A., 164, 250, 251
Biskup, J., 251
Bjorner, D., 172
Blasgen, M. W., 110, 172, 206
Bleier, R. E., 91
Bloco, 52, 53
Bloco de comunicação do programa (IMS), 256, 267
como esquema externo, 267
como responsável pela posição corrente, 282
definição dentro da aplicação, 270
Bloco de condição (QBE), 178
Bloco de especificação do programa (IMS), 256, 267
Bloqueio, 344
Boyce, R. F., 111, 224, 233
Brown, A. P. G., 438
- Buchholz, W., 71
Byte de preenchimento (HDAM, HIDAM), 286
- Cadeia (banco de dados em rede), 82
Cadeia (estrutura de armazenamento), 56
Cadiou, J. -M., 142, 251
Cálculo de domínio, 210, 216
Cálculo de tupla, 210
Cálculo relacional, 209
Caminho de acesso, 56
Caminho hierárquico (IMS), 267
Campo (IMS), 257
Campo (Sistema R), 114
Campo armazenado, 35
Campo de ordenação (IMS), 261
não-único, 261
Campo de subsequência (IMS), 339
Campo indexado (indexação secundária do IMS), 331
Campo lógico, 36
Campo sensitivo (IMS), 267
Campo virtual, 37
IMS, 315
Campo XD (IMS), 332
Campos relacionados ao sistema (IMS), 339
Campos superpostos (IMS), 262
Canning, R. G., 48
Cardenas, A. F., 49
Cardinalidade, 94
Carga do banco de dados (IMS), 268, 277
banco de dados lógico, 316
HDAM, 293-294
HIDAM, 296
HISAM, 287
HSAM, 286
MSDB, 344
Carlson, C. R., 438
Casey, R. G., 172
Catálogo, *veja* Dicionário
CC, *veja* COBOL Committee
Chamadas de caminhos (IMS), 278, 280
Chamberlin, D. D., 103, 110, 118, 154, 164
Chang, C. L., 142
Chang, H. K., 64, 73
Chang, P. Y. - T., 205
CHANGE (suboperação de FLD), 346
Chave
alternativa, 98
candidata, 97, 233
estrangeira, 99
primária, 54, 98
totalmente concatenada, *veja* Chave totalmente concatenada
Chave alternada, 98
Chave candidata, 97, 233
Chave concatenada, *veja* Chave totalmente concatenada
Chave de controle de ordenação (DBTG), 371
Chave do banco de dados (DBTG), 383

- Chave estrangeira, 99
Chave primária, 54, 97
Chave totalmente concatenada (IMS), 268
 indexação secundária, 334
Chaves candidatas superpostas, 233-236
CHECK (DBTG), 375
Chen, P. P. -S., 425, 439
Chiang, T. C., 74
Childs, D. L., 207
Classe de inserção (DBTG), 371
Classe de inserção MANUAL (DBTG), 373
Classe de membro (DBTG), 371
Classe de retenção (DBTG), 371
Classe de retenção FIXED (DBTG), 372
Classe de retenção MANDATORY, 373
Classe de retenção OPTIONAL (DBTG), 373
Clemons, E. K., 382
CLOSE (SQL embutida), 147
CNT (QBE), 182
COBOL Committee
 ANSI, 353
 CODASYL, *veja* CODASYL
COBOL *Journal of Development*, 352
CODASYL
 Comitê COBOL, 352
 Comitê de desenvolvimento, 207
 Comitê de linguagem de descrição de dados, 353
 Comitê de sistemas, 48, 50
Codd, E. F., 102, 103, 142, 164, 172, 203, 209, 219, 223, 224, 233, 437, 439
Codificação de dados, 36
Codificação de Huffman, 72
Código de tipo de segmento (IMS), 261
Códigos de comando (IMS), 278
 D, 278
 F, 280
 V, 280
Colisão (randomização), 61
 HDAM, 294
Comer, D., 71
Comitê de Linguagem de Descrição de Dados
 ANSI, 353
 CODASYL, *veja* CODASYL
COMMIT (DBTG), 400
COMMIT (UDL), 421
Comparações de desempenho, 423, 497-500
Compartilhamento (concorrência), 28
Compatível de união, 195
Completemente relacional, 202
Componente independente, 238
Compressão de dados, 69, 70, 72, 440-441
Compressão frontal, 68
Compressão posterior, 68
Condição de junção, 87, 122
Condição de membro (cálculo de domínio), 216
Conjunto (DBTG), 357
Conjunto agregado, 357
Conjunto agregado (UDL), 405
Conjunto CODASYL, 358
Conjunto DBTG, 357
Conjunto de banco de dados, 356
Conjunto de base (UDL), 405
Conjunto de dono acoplado, *veja* Conjunto (DBTG)
Conjunto de estruturas de dados, 357
Conjunto de registros (UDL), 405
Conjunto em seqüência (árvore-B), 64
Conjunto índice (árvore-B), 64
Conjunto potência, 468
Conjunto singular, 366
Conjunto vazio, 357
CONNECT (DBTG), 389
CONNECT (UDL), 421
Consistência, 32
Controle de uso, 46
Corrente (DBTG), 383
Corrente da unidade de corrida (DBTG), 385
COUNT (SQL), 132
CREATE INDEX (SQL), 114
CREATE TABLE (SQL), 113
Crescimento do banco de dados, 37, 78
 DBTG, 381, 486
 IMS, 265-266
 Sistema R, 160
Criação de tabela (QBE), 187
Crick, M. F., 172
Cuff, R. N., 192
*CUPID, 89, 191-192
Cursor (SQL embutida), 145
Cursor UDL, 405
Cursor limitado (UDL), 418
*CZAR, 48
Dados de interseção (IMS), 314
 variável, 328
Dados do usuário no índice (IMS), 339
Dados duplicados no índice (IMS), 340
Dados operacionais, 29
Dados secretos, 163
Dados variáveis de interseção, 328
*DAMAS, 220, 221
Data base, *veja* Bancos de dados
*dataBASIC, 48
Date, C. J., 142, 163, 423, 437
Dayal, U., 164, 251
DB - STATUS (DBTG), 386
DBA, *veja* Administrador do banco de dados
DBCS, 354
DBD, *veja* Descrição do banco de dados
DBD físico (IMS), 314
DBD lógico (IMS), 314
DBMS, 28, 45
*DBOMP, 48, 90
DBTG, 366
DBTG comparada à UDL, 414
DDL, 42
DDL conceitual, 43

- DDL externa, 42
 DDL interna, 44
DDL Journal of Development, 353
DDLC, *veja* Comitê da Linguagem de Descrição de Dados
 de Jong, S. P., 192
 de Maine, P. A. D., 72
 Deckert, K. L., 172
 Decomposição sem perdas, 231
 DEDB, 343
 *DEDUCE, 142, 210
 DEFINE VIEW (SQL), 156
 Definição de estrutura de memória, 44
 DELETE (SQL), 137
 embutida, 145
 DELETE CURRENT (SQL embutida), 147
 Dell'Orco, P., 142
 Delobel, C., 250, 252
 Dependência de múltiplos valores, 240, 241
 Dependência de junção, 242
 Dependência funcional, 224
 Dependência funcional total, 226
 Dependência transitiva, 231
 DESCRIBE (SQL dinâmica), 151
 Descrição do banco de dados (IMS), 255, 260
 DEDB, 349
 físico, 259, 299
 índice, 302, 334
 lógico, 314
 MSDB, 344
 Destruição de tabela (QBE), 188
 Determina funcionalmente, 225
 Determinante, 233
 Determinante funcional, 233
 Diagrama de Bachman, 358
 Diagrama de dependência, 226
 Diagrama de dependência funcional, 226
 Diagrama de estrutura de dados, 358
 DIAM, 73
 Dicionário, 47, 50
 QBE, 186
 Sistema R, 136, 156
 Dicionário de dados, 47, 49
 Diferença (álgebra relacional), 196
 Diretório (RSS), 172
 DISCONNECT (DBTG), 391
 DISCONNECT (UDL), 421
 *Disk Forte, 48
 Dispositivo de comunicação de dados, 254
 Dispositivos COBOL para bancos de dados, 353
 DISTINCT (UDL), 419, 493
 DIVIDE BY, 200
 Divisão (álgebra relacional), 200
 Divisão/resto (randomização), 59
 DK/NF, 247, 252
 DLET, *veja* Remoção (IMS)
 DML, 40
 *DMS-1100, 49, 90
- Dodd, G. G., 71, 355
 Dodd, M., 72
 Domínio, 78, 93
 QBE, 187
 Domínio primário, 96
 Domínio simples, 96
 Dono (DBTG), 357
 DROP INDEX (SQL), 115
 DROP TABLE (SQL), 114
 DROP VIEW (SQL), 157
 DSDL, 353, 378
 DSG, 297
 DSL, 40
 DSL ALPHA, 89, 209
 DUPLICATES (DBTG), 371
 Dzubak, B. J., 74
- Earnest, C. P., 438
 *EDMS, 49
 Elemento constante, 174
 Elemento de exemplo (QBE), 174
 Eliminação de duplicatas, 88
 QBE, 174, 182
 QUEL, 213
 SQL, 120, 132
 UDL, 419, 492
- Empresa, 29
 END TRANSACTION (SQL embutida), 150
 Endereçamento randômico, *veja* Randomização
 Endereço de registro armazenado, 53
 Engles, R. W., 29, 355, 491
 Entidade, 31
 Epstein, R., 219, 220
 Equijunção, *veja* junção
 ERASE (DBTG), 388
 Esquema (DBTG), 352, 353
 Esquema conceitual, 43, 425
 Esquema externo, 42
 Esquema interno, 44
 Essencialidade, 430
 Estabilidade (esquema conceitual), 425
 Estratégia de acesso, 34
 Estrutura de armazenamento hierárquica, 59
 Estrutura secundária de dados (IMS), 337
 Estructuras secundárias, 334
 Eswaran, K. P., 172, 206
 Everest, G. C., 49
 EXECUTE (SQL dinâmica), 151
 EXISTS (SQL), 128
 EXISTS (UDL), 412
 EXPAND TABLE (SQL), 114
 Expansão de tabela, 189
 Exploração simétrica, 429
 Explosão de peças
 DBTG, 490
 redes, 432
 relações, 432
 SQL, 452-453
 UDL, 422, 496-497

- Extensão (de uma relação), 99
 Fadous, R., 251
 Fagin, R., 75, 224, 242, 245, 247, 249, 250, 251,
 252
 Fato, 428
 FD, 224
 Fechamento (de linguagens relacionais), 193, 430
 Fehder, P. L., 73
 Feldman, J. A., 207
 FETCH (SQL embutida), 147
 Filho (IMS), 258, 259
 físico, 313
 lógico, *veja* Filho lógico
 Filho físico (IMS), 313
 Filho lógico (IMS), 314
 INDEX/HIDAM, 302
 indexação secundária, 306, 334
 FIND (DBTG), 392
 ANY, 392
 chave de identificação do banco de dados, 395
 DUPLICATE, 392
 DUPLICATE WITHIN USING, 395
 FIRST, 393
 LAST, 393
 NEXT, 393
 OWNER, 392
 PRIOR, 393
 RETAINING CURRENCY, 396
 WITHIN USING, 394
 FIND (UDL), 412, 420
 FINISH (DBTG), 400
 FLD (MSDB), 345
 Forma normal, 223
 domínio/chave, 247, 252
 prenex, 214
 primeira, 96, 223, 227
 projeção-junção, 245
 quarta, 241
 quinta, 244
 segunda, 229
 terceira, *veja* Terceira forma normal
 Forma normal de Boyce/Codd, 233
 Forma normal domínio/chave, 247, 252
 Forma normal prenex, 214
 Forma normal projeção/junção, 245
 Fórmula bem-formada, 211
 Forsyth, J., 251
 FORTRAN, 355
 *FQL, 219
 Fragmentação de memória, 301
 Fragmentação de memória (IMS), 301
 Frank, R. L., 355
 FREE (DBTG), 398
 FREE (UDL), 420
 Fry, J. P., 49
 FSA, 345
 Função, 95
 Função, *veja* Funções embutidas
- Função de randomização, 59
 Funções integradas
 QBE, 182
 Furtado, A. L., 164, 206
- *GAMMA-0, 172
 Gêmeo Físico (IMS), 314
 Gêmeo lógico (IMS), 313
 Gerritsen, R., 377
 GET (DBTG), 387
 GET (IMS), 273
 hold, 277
 next, 274
 next within parent, 276
 unique, 273
 GHN, *veja* GET (IMS)
 GHNP, *veja* GET (IMS)
 Ghosh, S. P., 71
 GHU, *veja* GET (IMS)
 *GIS, 48
 GN, *veja* GET (IMS)
 GNP, *veja* GET (IMS)
 Goldman, J., 204
 Goldstein, R. C., 204
 Goodman, N., 251
 Gotlieb, L. R., 206
 Gould, J. D., 191
 Grau (de uma relação), 93
 Gray, J. N., 164
 Greenblatt, D., 191
 GROUP BY (SQL), 134
 Grupamento (QBE), 183
 Grupo de arquivos, 297
 Grupo de arquivos primários (IMS), 297
 Grupo de arquivos secundários (IMS), 297
 Grupo de estudos do DBMS, 38
 Grupo repetitivo, 59
 não em relações, 102
 GSAM, 285
 GU, *veja* GET (IMS)
 GUIDE, 48
- Hall, P. A. V., 205
 Hammer, M. M., 111
 Harary, F., 73
 HAVING (SQL), 134
 Hawthorne, P., 220
 Heath, I. J., 206, 231, 241, 245
 Held, G. D., 219, 438
 HDAM, 291
 HIDAM, 291
 Hierarquia, 79
 HISAM, 287
 usando ISAM/OSAM, 287-290
 usando VSAM, 390-391
 HISAM simples, 286, 301
 Hitchcock, P., 205
 Hopewell, P., 142, 163
 Howard, J. H., 249

- HSAM, 285
 HSAM simples, 286, 301
 Hsiao, D. K., 73
 Huffman, D. A., 72
 Huits, M. H. H., 424
 Identificador de tupla (RSS), 167
 *IDS, 48, 49, 90, 91
 *IDMS, 49, 90
 *ILL, 219
 Implicação, 218
 Implicação lógica, 218
 Impressão (QBE), 174
 Imunidade ao crescimento, 161
 IN (SQL), 125
 Inconsistência, 32
 Independência de dados, 34-37, 50
 - física, 142, 161
 - IMS, 305, 340
 - lógica, 161
 - Sistema R, 140
 Independência de dados lógicos, 161
 Independência física de dados, 142, 161
 Indexação completa, 71, 443
 Indexação dispersa (IMS), 339
 Indicador de localização do pai físico (IMS), 321
 Indicador de localização do pai lógico (IMS), 314, 315
 Indicador de localização filho lógico (IMS), 321
 Indicador de localização gêmeo (IMS), 291
 - físico, 291
 - lógico, 323
 Indicador de localização gêmeo físico (IMS), 291
 Indicador de localização gêmeo lógico (IMS), 323
 Indicador de localização simbólico, 69
 - IMS, 315, 322, 334
 Indicadores de localização
 - banco de dados lógico, 311
 - DBTG, 358
 - DEDB, 348
 - direto, 69
 - HDAM/HIDAM, 291
 - HISAM, 287
 - indexação secundária, 332
 - simbólico, 69
 Indicadores de localização filho/gêmeo (IMS), 291
 Indicadores de localização hierárquicos (IMS), 291
 Índice
 - QBE, 187
 - Sistema R, 105-107, 168-169
 - Índice aglomerado (Sistema R), 169
 - Índice combinado, 67, 68, 71, 72
 - Índice de múltiplos níveis, 62, 71
 - Índice denso, 56
 - Índice duplo, 70
 - Índice estruturado em árvore, 62
 - Índice não-denso, 62
 - Índice secundário, 56
 - IMS, 330
 *INGRES, 90, 104, 219, 220
 Inserção (IMS), 276
 - bancos de dados lógicos, 317
 - DEDB, 349
 - HDAM, 294-296
 - HIDAM, 296
 - HISAM, 287-291
 - HSAM, 286
 - indexação secundária, 334
 - MSDB, 344
 Inserção (QBE), 185
 Inserção (SQL), 136
 - embutida, 145
 Instantâneo (QBE), 188
 Integridade, 33
 Integridade entidade, 98
 Integridade referencial, 99
 Intensão (de uma relação), 99
 Intercâmbio de dados, 33
 Interface de pesquisa à memória (Sistema R), 107, 165
 Interface do registro armazenado, 52, 53
 - DBTG, 377-378
 - IMS, 384
 - Sistema R, 165
 Interface do registro físico, 52, 61-66
 Interface do usuário, 47
 Interface voltado para o usuário (Sistema R), 107, 108
 Interseção (álgebra relacional), 195
 Inversão (QBE), *veja* Índice
 *IS/1, 204
 ISAM, 384
 *ISL-1, 48
 ISRT, *veja* Inserção (IMS)
 Item de dados (DBTG), 356
 Jardine, D. A., 49, 50
 JD, 244
 Jefferson, D. K., 438
 Junção (álgebra relacional), 87, 198
 - equijunção, 88, 198-199
 - externa, 203
 - generalizada, 242
 - maior do que, 198
 - natural, 198
 - pode ser, 203
 - SQL, 122
 Junção externa, *veja* Junção
 Junção maior do que, *veja* Junção
 Junção natural, *veja* Junção
 Junção pode ser, *veja* Junção
 Kapp, D., 256
 Kay, M. H., 382
 KEEP (DBTG), 399
 Kent, W., 252
 Kerschberg, L., 206
 Kim, W., 103

- King, W. F., III, 111
 Klug, A., 49
 Knuth, D. E., 64, 65, 71
 Kreps, P., 219
 Kroenke, D., 49
 Kuhns, J. L., 209, 219
 Lacroix, M., 207, 219
 Langefors, B., 50
 LAST (QBE), 181
 LCHILD, *veja* filho lógico
 LDB, 256, 265, 309
 LDBR, 265, 309
 *LEAP, 207
 Leben, J. F., 256
 Lefkowitz, D., 71
 LET (SQL embutida), 147
 Levein, R. E., 207
 Lewis, T.G., 71
 ligação (hierarquia), 80
 ligação (rede), 82
 ligação (RSS), 165
 *LILA, 142
 linguagem de consulta, 29
 linguagem de definição de dados, 42
 linguagem de descrição de dados, 42
 linguagem de descrição do armazenamento de dados (DBTG), 353, 377
 linguagem de especificação de acesso (Sistema R), 154
 linguagem de manipulação de dados, 42
 linguagem em camadas, 404, 474
 linguagem principal, 40
 - QBE, 173
 - SQL, 107, 143
 linguagem unificada de bancos de dados, 404, 474
 *LINUS, 142
 lista alvo, 211
 lista de materiais, *veja* explosão de peças
 lista de retenção (DBTG), 397, 488, 490
 Lochovsky, F. H., 49, 256
 Lohman, G. M., 74
 Lorie, R. A., 111, 154, 172
 Lum, V. Y., 72
 *MacAIMS, 204
 *MAGNUM, 49, 90, 104
 mapeamento, 40, 44
 Mapeamento (SQL: SELECT-FROM-WHERE), 119
 Mapeamento conceitual/interno, 44
 - DBTG, 377-378
 - IMS, 301
 - Sistema R, 165
 Mapeamento externo/conceitual, 40
 - DBTG, 379
 - IMS, 265
 - Sistema R, 155
 *MARK IV, 48, 90
 Maron, M. E., 207
 Marron, B. A., 72
 *MARS III, 48
 Martin, J., 71
 Materialização, 36
 Materialização dos dados, 36
 MATCHING (UDL), 419
 Maurer, W. D., 71
 MAX (QBE), 182
 MAX (SQL), 132
 McCreight, C., 64, 72
 McDonald, N., 191
 McGee, W. C., 49, 256, 438
 Membro (DBTG), 357
 Mercz, L. I., 382
 Merret, T. H., 206
 Metaxides, A., 438
 Método de acesso, 52
 Método de acesso à memória virtual (OS/VS), 64, 72
 Método de acesso direto hierárquico (IMS), 291
 Método de acesso indexado direto hierárquico (IMS), 291
 Método de acesso indexado seqüencial (OS/VS), 384
 Método de acesso indexado seqüencial hierárquico (IMS), 287
 Método de acesso seqüencial de excedentes (IMS), 384
 Método de acesso seqüencial generalizado (IMS), 385
 Método de acesso seqüencial hierárquico (IMS), 285
 Michaels, A. S., 438
 MIN (QBE), 182
 MIN (SQL), 132
 MINUS, 195
 Mittman, B., 438
 Modelo entidade relacionamento, 425
 Modelo de acesso independente de dados, 73
 Modelo de dados relacional, 202
 Modelo relacional, 202
 MODIFY (DBTG), 389
 Morris, R., 71
 *MRDS, 142
 MSDB, 343
 Mullin, J. K., 72
 Multidependente, 240
 Multidetermina, 240
 MVD, 240, 241
 Navegação, 402
 Nicolas, J. M., 242, 249
 Niebuhr, K. E., 191
 Nievergelt, J., 73
 Nijsen, S. M., 438, 439
 Nilsson, J.F., 154
 *NIPS/FFS, 48

- Nível conceitual, 38
 Nível externo, 38
 Nível interno, 38
 *NOMAD, 90, 104
 Nome funcional, 95
 Nomeação (de relações derivadas) QUEL, 214
 álgebra relacional, 196
 SQL, 156
 NONULL (SQL), 114
 Normalização, 96, 222, 468
 NOSET (UDL), 493
 Notley, M. G., 204
 Nulo, 96
 álgebra relacional, 202
 IMS, 278
 não na chave primária, 98
 QBE, 182
 QUEL, 215
 SQL, 113, 114, 131, 132, 135, 137, 154, 160, 433
 Objetivos da abordagem relacional, 437
 Olle, T. W., 355
 Opções de processamento (IMS), 268
 OPEN (SQL embutida), 147
 Operação de atualização, 79
 Operações primitivas (álgebra relacional), 203, 219
 *ORACLE, 140
 Ordenação
 atributos, 95
 controlada pelo programa, 94, 261
 controlada pelo valor, 94
 DBTG, 371, 401
 definido pelo sistema, 94
 essencial, 435
 IMS, *veja* seqüência de processamento
 primeiro a chegar/primeiro a sair, 94, 261
 QBE, 176
 SQL (CREATE INDEX), 114
 SQL (ORDER BY), 122
 tuplas, 94, 102, 474
 Ordenação controlada pelo programa, *veja*
 Ordenação
 Ordenação controlada pelo valor, *veja* Ordenação
 Ordenação definida pelo sistema, *veja* Ordenação
 Ordenação essencial, 435
 ORDER BY (SQL), *veja* Ordenação
 Organização comum, 58
 Organização de listas múltiplas, 58
 Organização invertida, 58
 OSAM, 384
 Ottimização (Sistema R), 108
 Ottimização (sistemas relacionais), 204, 205
 Pai (IMS)
 corrente, 276
 físico, 313
 lógico, 313
 Pai corrente (IMS), 276
 Pai físico (IMS), 313
 Pai lógico (IMS), 313
 Página (Sistema R), 166
 Palermo, F. P., 205
 Palmer, I. R., 49
 Parâmetros (SQL dinâmica), 152
 Parelha (IMS), 319
 física, 319
 virtual, 321
 Parelha física, 319
 Parelha virtual, 321
 Parker, D. S., 250, 252
 PCB, *veja* Bloco de comunicação do programa
 PDB, 254, 257
 PDBR, 257
 Pecherer, R. M., 206
 Peterson, W.W., 71
 *PHOLAS, 377
 Pippenger, N., 75
 Pirotte, A., 207, 219
 PJ/NF, 244
 Polya, G., 439
 Ponto de sincronização ou ponto de parada, 344
 Ponto de sincronização (IMS), 344
 POS (DEDB), 349
 Posição corrente (IMS), 274, 280, 282
 get next, 274
 insert, 276-277
 Possibilidade de atualização de visões, 158, 163
 Pré-compilador (Sistema R), 108
 Predicado (SQL), 121
 Predicado (RSS), 170
 Prefixo (IMS), 286, 307
 Prefixo (Sistema R), 167
 PREPARE (SQL dinâmica), 150
 Price, T. G., 154
 Primeiro a chegar/primeiro a sair, *veja* Ordenação
 Primeira forma normal, 96, 223, 227
 PROCOPT, 268
 Produto Cartesiano, 94
 Produto Cartesiano estendido (álgebra relacional), 195
 programa on-line, 29
 Programador de aplicações, 29
 Projeção independente, 237
 PROJECT (álgebra relacional), 87
 Propagação de atualizações, 32
 *PRTV, 140, 204, 205
 PSB, 256, 267
 QBE, 89, 90, 104, 183
 Qualificação do cursor (UDL), 412
 Quantificador, 212
 Quantificador existencial, 212
 Quantificador universal, 212
 Quarta forma normal, 241
 QUEL, 89, 213-216

Query By Example, 89, 90, 104, 173
Quinta forma normal, 244

Raiz, 80
*RAM, 172
Randomização, 59
 extensível, 75

Randomização estendida, 75
RANGE (QUEL), 213

*RDMS, 204

RDS, 107

READY (DBTG), 381, 400

Realm (DBTG), 380

RECONNECT (DBTG), 391

RECONNECT (UDL), 421

Recuperação, 46

Réde de *n* entidades, 324, 364

Redução à forma normalizada, 96, 222, 468

Redundância, 32, 222, 428

 controlada, 32, 428

 DBTG, 368

 forte, 470

 IMS, *veja* Dados duplicados; parelha física
 relações, 429

Redundância controlada, 32

 dados duplicados (IMS), 339

 parelha física (IMS), 321

Redundância forte, *veja* Redundância

Reestruturação do banco de dados, 161

Referência a cursor qualificado (UDL), 412

Referência direta (UDL), 412

Registro armazenado, 35

 RSS, 167

Registro conceitual, 43

Registro de banco de dados físico (IMS), 257

Registro externo, 42

Registro físico, 52, 53

Registro interno, 43

Registro lógico, 35, 42

 OS/VS, 285

Registro lógico de banco de dados (IMS), 263,
 309

Regras (IMS)

 ISRT/DLET/REPL, 317

 relacionamentos lógicos, 327

Regras de inserção (IMS), 317

Regra de Integridade 1, 98

Regra de Integridade 2, 98

Regras de remoção (IMS), 318

Regras de substituição (IMS), 317

Reisner, P., 111

Relação, 77, 93, 94, 102

 UDL, 405

Relação atômica, 238

Relação binária, 429

Relação estruturada em árvore, 179

Relação funcional, 250

Relação *n*-ária, *veja* Relação

Relação não-normalizada, 96

Relação normalizada, 96

Relação quociente, 206

Relação unária, 94

Relação universal, 252

Relacionamento, 30, 31, 89

Relacionamento bidirecional (IMS), 319

Relacionamento lógico (IMS), 256, 265, 309

Relacionamento unidirecional (IMS), 319

Relações *versus* arquivos, 102

Remoção (IMS), 277

 banco de dados lógico, 317

 DEDB, 349

 HDAM, 296

 HIDAM, 297-291

 HSAM, 286

 indexação secundária, 334

 MSDB, 344

Remoção (QBE), 185

*RENDEZVOUS, 142

Reorganização, 46, 54, 69

 IMS, 304

REPL, *veja* Substituição (IMS)

Representação abstrata, 38

Representação conforme, 142

Restrição de chave (DK/NF), 247

Restrição de domínio (DK/NF), 247

Restrição inter-relacional, 237

Restrições estruturais (DBTG), 375

RETAINING CURRENCY (DBTG), 396

Reversibilidade do processo de redução, 229, 231

Richardson, J. S., 72

Rissanen, J., 237, 238, 242, 250, 251

Robinson, K. A., 439

ROLB (IMS), 348

ROLLBACK (DBTG), 400

ROLLBACK (UDL), 421

Ross, R. G., 49

Rothnie, J. B., Jr., 220

Roussopoulos, N., 142

Rovner, P. D., 207

RSI, 107, 165

RSS, 107, 165

Rustin, R., 437

Sagiv, Y., 252

*SC-1, 48

Schenk, H., 377

Schmid, H. A., 240, 251

Segmento (IMS), 257

Segmento (Sistema R), 113, 166

 privativo, 113, 166

 público, 113, 166

 temporário, 166

Segmento alvo do índice (IMS), 334

Segmento concatenado (IMS), 314

Segmento corrente (IMS), *veja* Posição corrente

Segmento dependente (IMS), 258, 259

Segmento dependente seqüencial (DEDB), 348

- Segmento fonte do índice (IMS), 340
Segmento gêmeo (IMS), 250
 físico, 259, 314
 lógico, 314
Segmento indicador de localização (IMS), 313
Segmento índice (IMS), 297, 334
Segmento parelha (IMS), 319
Segmento prefixo (IMS), 286, 307
Segmento privativo (Sistema R), 113
Segmento público (Sistema R), 113
Segmento raiz (IMS), 258
 banco de dados lógico, 327
 indexação secundária, 337
Segmento sensível (IMS), 265
Segmento virtual (IMS), 322
Segmentos diretamente dependentes (DEDB), 348
Segunda forma normal, 229
Segurança, 33
 IMS, 266
SELECT (álgebra relacional), 88
SELECT (SQL), 119
SELECT de tabela com uma só linha (SQL embutida), 145
Seletividade (índice), 67
 IMS, 339
Selinger, P. G., 154
Semi-relacional, 203
Senko, M. E., 73, 191
Sensibilidade de chave (IMS), 268
Sensibilidade de campo (IMS), 267, 269
SEQUEL, 105
SEQUEL/2, 111, 142
Seqüência (UDL), 405
Seqüência de processamento (IMS)
 primária, 333
 secundária, 333
Seqüência do sistema (RSS), 107, 165, 168
Seqüência hierárquica (IMS), 262
Seqüência primária de processamento (IMS), 333
Seqüência secundária de processamento (IMS), 333
Ser completo, *veja* Ser relationalmente completo
Ser relationalmente completo, 132, 193, 202,
 218
 SQL, 203, 459-460
SERIES, 48
SET SELECTION (DBTG), 374
Sevcik, K. C., 164
Severance, D. G., 73, 74
SHARE, 48, 423
SHISAM, 286, 301
Shneiderman, B., 72
SHSAM, 286, 301
Sibley, E. H., 49, 207, 437
Simetria, 427, 428
Síntese (relações 3NF), 250, 251
Sistema de banco de dados, 26, 27
Sistema de controle de banco de dados (DBTG), 354
Sistema de controle em tempo de execução (Sistema R), 109
Sistema de dados relacional (Sistema R), 108
Sistema de gerenciamento de banco de dados, 29,
 45
Sistema de múltiplos usuários, 28
Sistema de pesquisa à memória (Sistema R), 101,
 165
Sistema on-line, 342
Sistema R, 163
Smith, J. M., 205, 247, 252
Smith, S. E., 191
Spadavecchia, V. N., 142
*SQUARE, 89, 105
*SQUIRAL, 205
SQL, 132
SQL embutida, 105, 143
*SQL/DS, 15
SRA, 54
SSA, 272-278
Status (IMS), 27
*STDS, 207
Steel, T. B., Jr., 439
Stewart, J., 204
Stonebraker, M. R., 50, 191, 219, 220, 438
STORE (DBTG), 387
Strnad, A. J., 204
Strong, H. R., 75
Subconsulta (SQL), 124
Subesquema (DBTG), 352, 353
Sublinguagem ALPHA de dados, 89, 209
Sublinguagem de dados, 40
Subsistema de armazenamento, 52
Substituição (IMS), 277
 banco de dados lógico, 317
 DEDB, 349
 indexação secundária, 334
 MSDB, 344
SUM (QBE), 182
SUM (SQL), 132
Sundgren, B., 50
Swenson, J. R., 240, 251
Symonds, A. J., 172
SYNC (IMS), 344
*System 2000, 49, 90, 256
Tabela, 77, 105
Tabela básica (Sistema R), 106-107, 112
Tabela virtual, 155
Taylor, R. W., 355
*TDMS, 48, 90, 91
Técnicas de compressão, 68, 69, 70, 72, 440-441
Terceira forma normal, 232
 definição intuitiva, 227
Thomas, J. C., 191
TID, 167
TIMES, 195

- Tipo de dado
 DBTG, 482
 IMS, 262
 IMS (fast Path), 343
 QBE, 187
 SQL, 113
 SQL (visões), 158
 Titman, P. J., 207
 Todd, S. J. P., 164, 204, 205
 *TOTAL, 48, 90
 Traiger, I. L., 164, 172
 *TRAMP, 207
 Transações, 150, 342
 Tsichritzis, D. C., 49, 256, 437
 Tupla, 78, 94, 101
 Tupla concatenada, 195
 Tupla n , 78, 94, 101
 UDL, 404, 421, 474
 UDL comparada à DBTG, 414
 UFI, 107, 108
 Uhrowczik, P. P., 50
 *UL/I, 48
 Ullman, J. D., 206, 219, 242
 União (álgebra relacional), 195
 Unidade de corrida, 384
 UNION (SQL), 130
 UNIQUE (índice do Sistema R), 115
 UNIQUE (SQL SELECT), 120
 UNQ (QBE), 182
 UPDATE (SQL), 135
 embutida, 145
 UPDATE CURRENT (SQL embutida), 147
 USAGE-MODE (DBTG), 400
 USE FOR DB-EXCEPTION (DBTG), 386
 Usuário, 29, 40
 Usuário eventual, 142
 Usuário final, 29
 Utilitário de carga (Sistema R), 113
 Utilitários, 46
 UWA, 353, 379
 Validação, 33
 Valor default (DBTG), 371
 Variável domínio, 210, 216
 Variável indicador (SQL embutida), 453
 Variável limitada, 211-213
 Variável livre, 211-213
 Variável tupla, 209, 210
 SQL, 210
 Varredura, 169
 VERIFY (suboperação de FLD), 345
 Visão (Sistema R), 105-106, 155
 Visão conceitual, 42
 Visão externa, 42
 Visão interna, 43
 Vorhaus, A. H., 91
 Vose, M. R., 72
 VSAM, 64, 72
 Wade, B. W., 154
 Wagner, R. E., 72
 Warburton, C. R., 74
 Waxman, J., 191
 WFF, 211
 Wiederhold, G., 71
 Williams, S. B., 91
 Wodon, P., 219
 Wong, E., 74, 219, 220
 X3H2, 353
 X3J4, 353
 XRDI, 109
 XRM, 172
 Yormark, B., 50
 Youssefi, K., 220
 Yuen, P. S. T., 72
 Zaniolo, C. A., 382
 Zloof, M. M., 173, 191, 192

Impressão e acabamento
(com filmes fornecidos);
EDITORIA SANTUÁRIO
Fone (0125) 36-2140
APARECIDA - SP