

Elmasri • Navathe

Sistemas de banco de dados

6^a Edição



Companion
Website

© 2011 by Pearson Education do Brasil.
© 2011, 2007, 2004, 2000, 1994 e 1989 Pearson Education, Inc.

Tradução autorizada a partir da edição original, em inglês, *Fundamentals of Database Systems*, 6th edition, publicada pela Pearson Education, Inc., sob o selo Addison-Wesley.

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

Diretor editorial: Roger Trimer
Gerente editorial: Sabrina Cairo
Supervisor de produção editorial: Marcelo Françozo
Editora plena: Thelma Babaoka
Editora de texto: Sabrina Levensteinas
Preparação: Paula Brandão Perez Mendes
Revisão: Elisa Andrade Buzzo
Capa: Thyago Santos sobre o projeto original de Lou Gibbs/Getty Images
Projeto gráfico e diagramação: Globaltec Artes Gráficas Ltda.

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

Elmasri, Ramez

Sistemas de banco de dados / Ramez Elmasri e Shamkant B. Navathe ; tradução Daniel Vieira ; revisão técnica Enzo Seraphim e Thatyana de Faria Piola Seraphim. -- 6. ed. -- São Paulo : Pearson Addison Wesley, 2011.

Título original: *Fundamentals of database systems*.

ISBN 978-85-7936-085-5

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

10-11462

CDD-005.75

Índices para catálogo sistemático:

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

2010

Direitos exclusivos para a língua portuguesa cedidos à

Pearson Education do Brasil,

uma empresa do grupo Pearson Education

Rua Nelson Francisco, 26 – Limão

Cep: 02712-100 São Paulo – SP

Tel: (11) 2178-8686 – Fax: (11) 2178-8688

e-mail: vendas@pearson.com

Estruturas de indexação para arquivos

Neste capítulo, consideramos que um arquivo já existe com alguma organização primária, como as organizações desordenada, ordenada ou hashed, que foram descritas no Capítulo 17. Vamos descrever outras estruturas de acesso auxiliares, chamadas índices, que são utilizadas para agilizar a recuperação de registros em resposta a certas condições de pesquisa. As estruturas de índice são arquivos adicionais no disco que oferecem caminhos de acesso secundários, os quais oferecem formas alternativas de acessar os registros sem afetar seu posicionamento físico no arquivo de dados primário no disco. Elas permitem o acesso eficiente aos registros com base nos campos de indexação que são usados para construir o índice. Basicamente, qualquer campo do arquivo pode servir para criar um índice, e múltiplos índices em diferentes campos — bem como índices em múltiplos campos — podem ser construídos no mesmo arquivo. Vários índices são possíveis; cada um deles utiliza determinada estrutura de dados para agilizar a pesquisa. Para encontrar um registro ou registros no arquivo de dados com base em uma condição de pesquisa em um campo de índice, o índice é pesquisado, o que leva a ponteiros para um ou mais blocos de disco no arquivo de dados onde os registros exigidos estão localizados. Os tipos mais predominantes de índices são baseados em arquivos ordenados (índices de único nível) e estruturas de dados em árvore (índices multinível, B'-trees). Os índices também podem ser construídos com base no hashing ou em outras estruturas de dados de pesquisa. Também vamos abordar os índices que são vetores de bits, chamados índices bitmap.

Descrevemos diferentes tipos de índices ordenados de único nível — primários, secundários e agrupamento — na Seção 18.1. Ao visualizar um índice de único nível como um arquivo ordenado, pode-

-se desenvolver índices adicionais para ele, fazendo surgir o conceito de índices multiníveis. Um esquema de indexação popular, chamado ISAM (*Indexed Sequential Access Method*) é baseado nessa ideia. Discutimos os índices multiníveis estruturados em árvore na Seção 18.2. Na Seção 18.3, descrevemos as B-trees e as B'-trees, que são estruturas de dados normalmente usadas em SGBDs para implementar dinamicamente índices multiníveis mutáveis. As B'-trees se tornaram uma estrutura padrão comumente aceita para a geração de índices por demanda na maioria dos SGBDs relacionais. A Seção 18.4 é dedicada a maneiras alternativas de acessar dados com base em uma combinação de múltiplas chaves. Na Seção 18.5, discutimos os índices de hash e apresentamos o conceito de índices lógicos, que dão um nível adicional de indireção dos índices físicos, permitindo que o índice físico seja flexível e extensível em sua organização. Na Seção 18.6, discutimos a indexação de chaves múltiplas e os índices bitmap usados para pesquisar uma ou mais chaves. No final do capítulo há um resumo.

18.1 Tipos de índices ordenados de único nível

A ideia por trás de um índice ordenado é semelhante à que está por trás do índice usado em um livro, que lista termos importantes ao final, em ordem alfabética, junto com uma lista dos números de página onde o termo aparece no livro. Podemos pesquisar o índice do livro em busca de certo termo em seu interior e encontrar uma lista de *endereços* — números de página, nesse caso — e usar esses endereços para localizar as páginas especificadas primeiro e depois *procurar* o termo em cada página citada. A alternativa, se nenhu-

ma outra indicação for dada, seria folhear lentamente o livro inteiro, palavra por palavra, para encontrar o termo em que estamos interessados. Isso corresponde a fazer uma *pesquisa linear*, que varre o arquivo inteiro. Naturalmente, a maioria dos livros possui informações adicionais, como títulos de capítulo e seção, que nos ajudam a localizar um termo sem ter de folhear o livro inteiro. No entanto, o índice é a única indicação exata das páginas onde o termo ocorre no livro.

Para um arquivo com determinada estrutura de registro consistindo em vários campos (ou atributos), uma estrutura de acesso a índice normalmente é definida em um único campo de um arquivo, chamado **campo de índice** (ou **atributo de indexação**).¹ O índice costuma armazenar cada valor do campo de índice junto com uma lista de ponteiros para todos os blocos de disco que contêm registros com esse valor de campo. Os valores no índice são *ordenados* de modo que possamos realizar uma *pesquisa binária* no índice. Se tanto o arquivo de dados quanto o arquivo de índice estiverem ordenados, e visto que este normalmente é muito menor do que o arquivo de dados, a procura no índice que usa pesquisa binária é uma opção melhor. Índices multiníveis estruturados em árvore (ver Seção 18.2) implementam uma extensão da ideia de pesquisa binária, que reduz o espaço de pesquisa pelo particionamento duplo em cada etapa de pesquisa, criando assim uma técnica mais eficiente, que divide o espaço de pesquisa no arquivo em n maneiras a cada estágio.

Existem vários tipos de índices ordenados. Um índice primário é especificado no *campo de chave de ordenação* de um arquivo ordenado de registros. Lembre-se, da Seção 17.7, que um campo de chave de ordenação é usado para *ordenar fisicamente* os registros de arquivo no disco, e cada registro tem um *valor único* para esse campo. Se o campo de ordenação não for um campo de chave — ou seja, se diversos registros no arquivo puderem ter o mesmo valor para o campo de ordenação —, outro tipo de índice, chamado índice de agrupamento (*clustering*), pode ser utilizado. O arquivo de dados é chamado de arquivo agrupado nesse último caso. Observe que um arquivo pode ter no máximo um campo de ordenação físico, de modo que pode ter no máximo um índice primário ou um índice de agrupamento, *mas não ambos*. Um terceiro tipo de índice, chamado índice secundário, pode ser especificado em qualquer campo *não ordenado* de um arquivo. Um arquivo de dados pode ter vários índices

secundários além de seu método de acesso primário. Discutimos esses tipos de índices de único nível nas três próximas subseções.

18.1.1 Índices primários

Um índice primário é um arquivo ordenado cujos registros são de tamanho fixo com dois campos, e ele atua como uma estrutura de acesso para procurar e acessar de modo eficiente os registros de dados em um arquivo. O primeiro campo é do mesmo tipo de dado do campo de chave de ordenação — chamado de **chave primária** — do arquivo de dados, e o segundo campo é um ponteiro para um bloco de disco (um endereço de bloco). Existe uma entrada de índice (ou registro de índice) no arquivo de índice para cada *bloco* no arquivo de dados. Cada entrada de índice tem o valor do campo de chave primária para o *primeiro* registro em um bloco e um ponteiro para esse bloco como seus dois valores de campo. Vamos nos referir aos dois valores de campo da entrada de índice i como $\langle K(i), P(i) \rangle$.

Para criar um índice primário no arquivo ordenado mostrado na Figura 17.7, usamos o campo Nome como chave primária, pois esse é o campo de chave de ordenação do arquivo (supondo que cada valor de Nome seja exclusivo). Cada entrada no índice tem um valor de Nome e um ponteiro. As três primeiras entradas de índice são as seguintes:

$\langle K(1) = (\text{Aaron}, \text{Eduardo}), P(1) = \text{endereço de bloco } 1 \rangle$

$\langle K(2) = (\text{Adams}, \text{João}), P(2) = \text{endereço de bloco } 2 \rangle$

$\langle K(3) = (\text{Alexandre}, \text{Eduardo}), P(3) = \text{endereço de bloco } 3 \rangle$

A Figura 18.1 ilustra esse índice primário. O número total de entradas no índice é igual ao *número de blocos de disco* no arquivo de dados ordenado. O primeiro registro em cada bloco do arquivo de dados é chamado de *registro de âncora do bloco* ou, simplesmente, *âncora de bloco*.²

Os índices também podem ser caracterizados como densos ou esparsos. Um índice denso tem uma entrada de índice para *cada valor de chave de pesquisa* (e, portanto, cada registro) no arquivo de dados. Um índice esparsão (ou não denso), por sua vez, tem entradas de índice para somente alguns dos valores de pesquisa. Um índice esparsão tem menos entradas do que o número de registros no arquivo. Assim, um índice primário é um índice não denso (esparsão), pois inclui uma entrada para cada bloco de disco do ar-

¹ Usamos os termos *campo* e *atributo* para indicar a mesma coisa neste capítulo.

² Podemos usar um esquema semelhante ao que foi descrito aqui, com o último registro em cada bloco (em vez do primeiro) como a âncora de bloco. Isso melhora ligeiramente a eficiência do algoritmo de pesquisa.

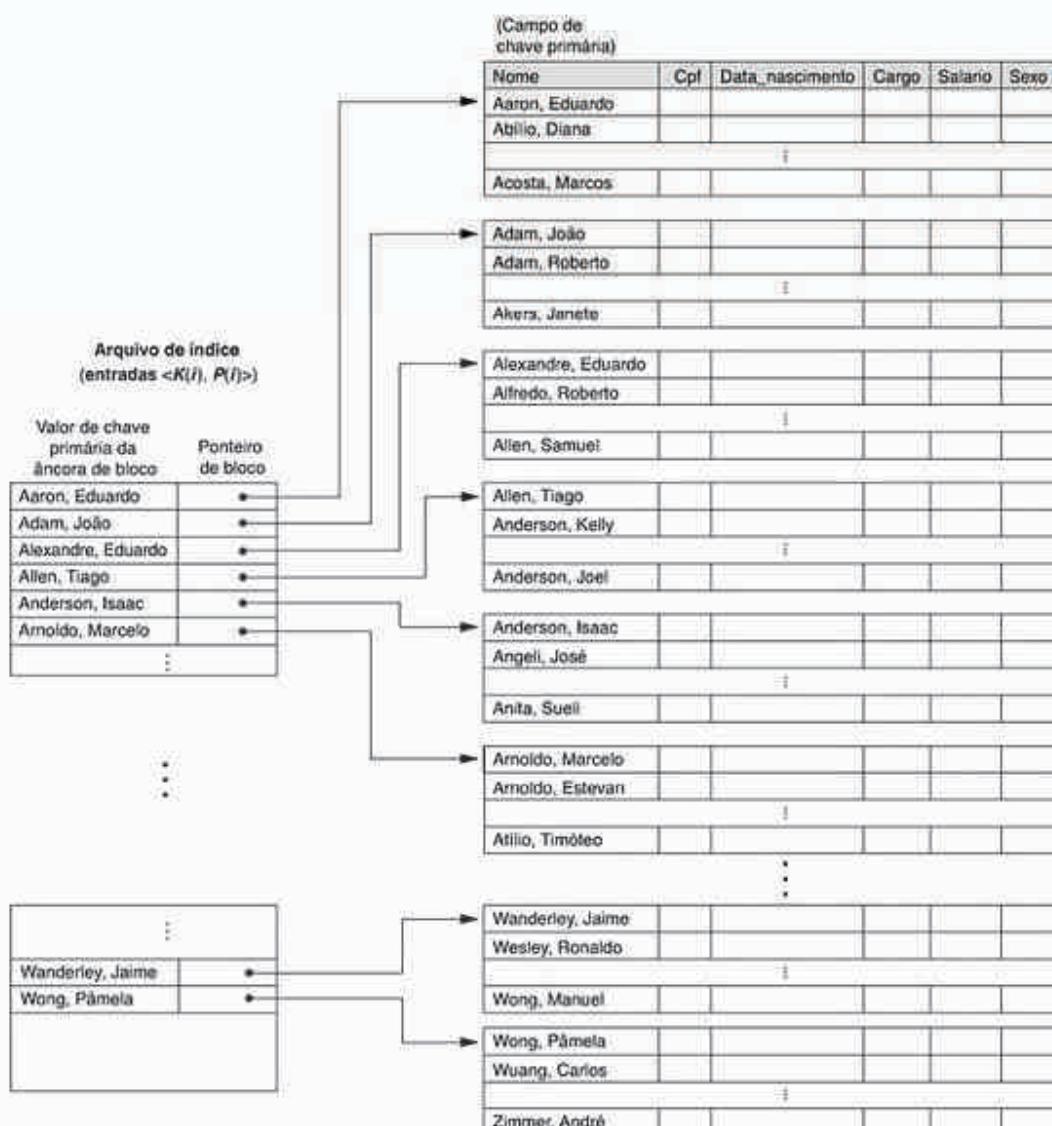


Figura 18.1
Índice primário no campo de chave de ordenação do arquivo mostrado na Figura 17.7.

quivo de dados e as chaves de seu registro de âncora, em vez de cada valor de pesquisa (ou cada registro).

O arquivo de índice para um índice primário ocupa um espaço muito menor do que o arquivo de dados, por dois motivos. Primeiro, existem *menos entradas de índice* do que registros no arquivo de dados. Segundo, cada entrada de índice normalmente é *menor em tamanho* do que um registro de dados, pois tem apenas dois campos; em consequência, mais entradas de índice do que registros de dados podem caber em um bloco. Portanto, uma pesquisa binária no arquivo de índice requer menos acessos de bloco do que uma pesquisa binária no arquivo de dados. Com relação à Tabela 17.2, observe

que a pesquisa binária para um arquivo de dados ordenado exigia $\log_2 b$ acessos de bloco. Mas, se o arquivo de índice primário tiver apenas b_i blocos, então localizar um registro com um valor de chave de pesquisa exige uma pesquisa binária desse índice e o acesso ao bloco que contém esse registro; um total de $\log_2 b_i + 1$ acessos.

Um registro cujo valor da chave primária é K se encontra no bloco cujo endereço é $P(i)$, onde $K(i) \leq K < K(i+1)$. O i -ésimo bloco no arquivo de dados contém todos os registros por causa da ordenação física dos registros do arquivo no campo de chave primária. Para recuperar um registro, dado o valor K de seu campo de chave primária, realizamos uma pesquisa binária no arquivo

de índice para encontrar a entrada de índice apropriada i , e depois recuperamos o bloco do arquivo de dados cujo endereço é $P(i)$.³ O Exemplo 1 ilustra a economia em acessos a bloco que pode ser alcançada quando um índice primário é utilizado para procurar um registro.

Exemplo 1. Suponha que tenhamos um arquivo ordenado com $r = 30.000$ registros armazenados em um disco com tamanho de bloco $B = 1.024$ bytes. Os registros de arquivo são de tamanho fixo e não espalhados, com tamanho de registro $R = 100$ bytes. O fator de bloco para o arquivo seria $bfr = \lceil(B/R)\rceil = \lceil(1.024/100)\rceil = 10$ registros por bloco. O número de blocos necessários para o arquivo é $b = \lceil(r/bfr)\rceil = \lceil(30.000/10)\rceil = 3.000$ blocos. Uma pesquisa binária no arquivo de dados precisaria de aproximadamente $\lceil\log_2 b\rceil = \lceil\log_2 3.000\rceil = 12$ acessos de bloco.

Agora, suponha que o campo de chave de ordenação do arquivo seja $V = 9$ bytes de extensão, um ponteiro de bloco seja $P = 6$ bytes de extensão e tenhamos construído um índice primário para o arquivo. O tamanho de cada entrada de índice é $R_i = (9 + 6) = 15$ bytes, de modo que o fator de bloco para o índice é $bfr_i = \lceil(B/R_i)\rceil = \lceil(1.024/15)\rceil = 68$ entradas por bloco. O número total de entradas de índice r_i é igual ao número de blocos no arquivo de dados, que é 3.000. O número de blocos de índice é, portanto, $b_i = \lceil(r_i/bfr_i)\rceil = \lceil(3.000/68)\rceil = 45$ blocos. Para realizar uma pesquisa binária no arquivo de índice, seriam necessários $\lceil(\log_2 b_i)\rceil = \lceil(\log_2 45)\rceil = 6$ acessos de bloco. Para procurar um registro usando o índice, precisamos de um acesso de bloco adicional ao arquivo de dados, para um total de $6 + 1 = 7$ acessos de bloco — uma melhoria em relação à pesquisa binária no arquivo de dados, que exigiu 12 acessos a bloco de disco.

Um problema importante com índice primário — assim como com qualquer arquivo ordenado — é a inserção e exclusão de registros. Com um índice primário, o problema é aumentado porque, se tentarmos inserir um registro em sua posição correta no arquivo de dados, temos de não apenas mover registros para criar espaço para o novo registro, mas também mudar algumas entradas de índice, pois a movimentação de registros mudará os *registros de âncora* de alguns blocos. Ao usar um arquivo de overflow desordenado, conforme discutimos na Seção 17.7, podemos reduzir esse problema. Outra possibilidade é usar uma lista ligada de registros de overflow para cada bloco no arquivo de dados. Isso é semelhante ao método de tratar de registros de overflow descrito com hashing na Seção 17.8.2. Os registros em cada bloco e sua lista ligada de overflow podem ser classificados para

melhorar o tempo de recuperação. A exclusão de registro é tratada com marcadores de exclusão.

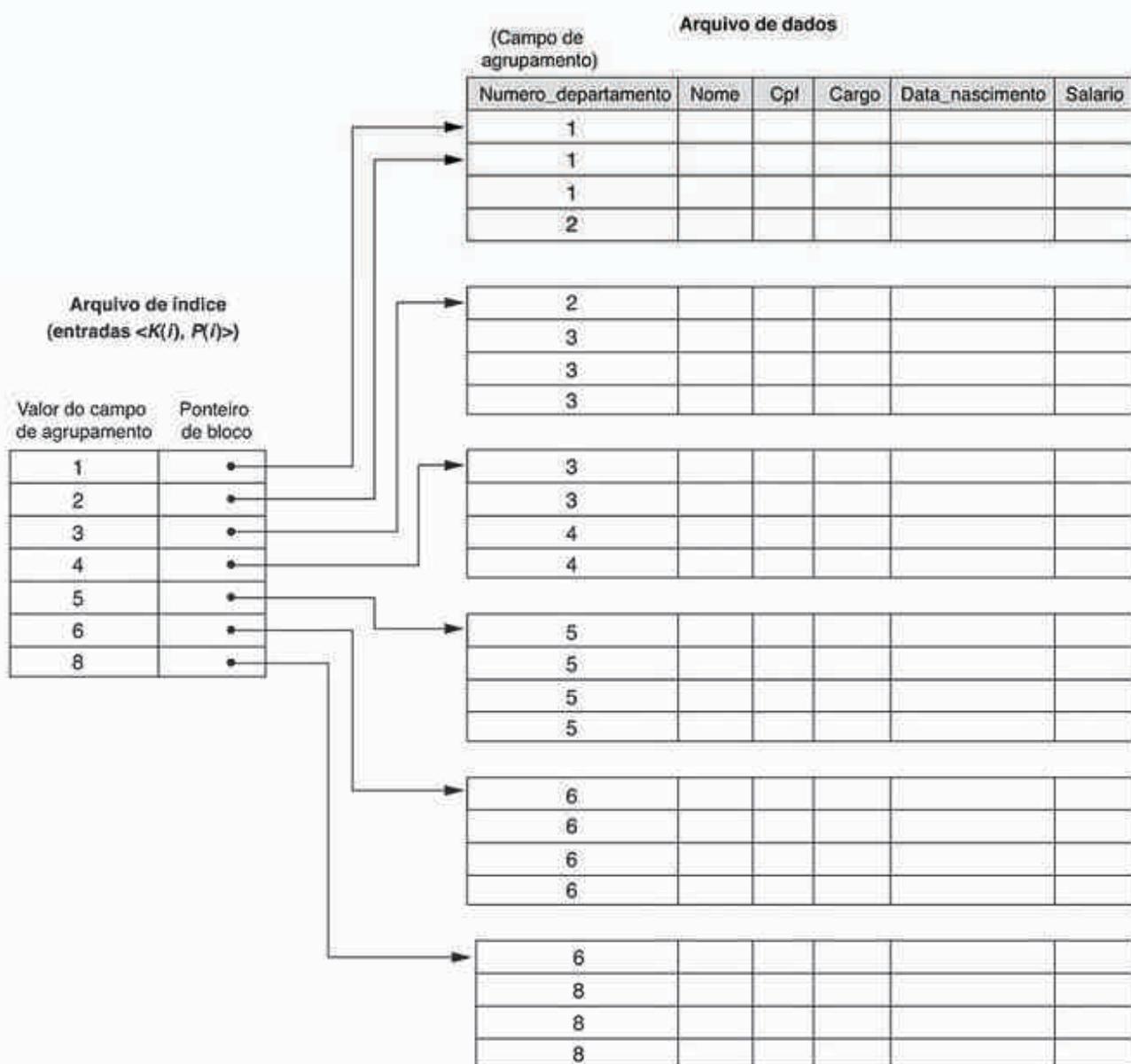
18.1.2 Índices de agrupamento

Se os registros de arquivo forem fisicamente ordenados em um campo não chave — que não tem um valor distinto para cada registro —, esse campo é chamado de *campo de agrupamento*, e o arquivo de dados é chamado de *arquivo agrupado*. Podemos criar um tipo de índice diferente, chamado *índice de agrupamento*, para agilizar a recuperação de todos os registros que têm o mesmo valor para o campo de agrupamento. Isso difere de um índice primário, que exige que o campo de ordenação do arquivo de dados tenha um *valor distinto* para cada registro.

Um índice de agrupamento também é um arquivo ordenado com dois campos; o primeiro campo é do mesmo tipo do campo de agrupamento do arquivo de dados, e o segundo campo é um ponteiro de bloco de disco. Há uma entrada no índice de agrupamento para cada *valor distinto* do campo de agrupamento, e ele contém o valor e um ponteiro para o *primeiro bloco* no arquivo de dados que tem um registro com esse valor para seu campo de agrupamento. A Figura 18.2 mostra um exemplo. Observe que a inserção e exclusão de registro ainda causam problemas, pois os registros de dados estão fisicamente ordenados. Para aliviar o problema de inserção, é comum reservar um bloco inteiro (ou um cluster de blocos contínuos) para *cada valor* do campo de agrupamento; todos os registros com esse valor são colocados no bloco (ou cluster de bloco). Isso torna a inserção e exclusão relativamente simples. A Figura 18.3 mostra esse esquema.

Um índice de agrupamento é outro exemplo de um índice *não denso*, pois ele tem uma entrada para cada *valor distinto* do campo de índice, que é uma não chave por definição e, portanto, tem valores duplicados em vez de um valor único para cada registro no arquivo. Existe alguma semelhança entre as figuras 18.1, 18.2 e 18.3 e as figuras 17.11 e 17.12. Um índice é de alguma forma semelhante ao hashing dinâmico (descrito na Seção 17.8.3) e às estruturas de diretório usadas para o hashing extensível. Ambos são pesquisados para encontrar um ponteiro para o bloco de dados que contém o registro desejado. Uma diferença principal é que uma pesquisa de índice usa os valores do próprio campo de pesquisa, enquanto uma pesquisa de diretório de hash usa o valor de hash binário que é calculado pela aplicação da função de hash ao campo de pesquisa.

³Observe que a fórmula dada não seria correta se o arquivo de dados fosse ordenado por um campo não chave; nesse caso, o mesmo valor de índice na âncora de bloco poderia ser repetido nos últimos registros do bloco anterior.

**Figura 18.2**

Um índice de agrupamento no campo não chave de ordenação `Numero_departamento` de um arquivo FUNCIONARIO.

18.1.3 Índices secundários

Um índice secundário oferece um meio secundário para acessar um arquivo de dados para o qual algum acesso primário já existe. Os registros do arquivo de dados poderiam ser ordenados, desordenados ou hashed. O índice secundário pode ser criado em um campo que é uma chave candidata e tem um valor único em cada registro, ou em um campo não chave com valores duplicados. O índice é novamente um arquivo ordenado com dois campos. O primeiro campo é do mesmo tipo de dado de algum campo não ordenado do arquivo de dados que seja um campo de índice. O segundo campo é um ponteiro

de bloco ou um ponteiro de registro. Muitos índices secundários (e, portanto, campos de indexação) podem ser criados para o mesmo arquivo — cada um representa um meio adicional de acessar esse arquivo com base em algum campo específico.

Primeiro, consideramos uma estrutura de acesso de índice secundário em um campo de chave (único) que tem um *valor distinto* para cada registro. Tal campo às vezes é chamado de chave secundária. No modelo relacional, isso corresponderia a qualquer atributo de chave UNIQUE ou ao atributo de chave primária de uma tabela. Nesse caso, existe uma entrada de índice para *cada registro* no arquivo de da-

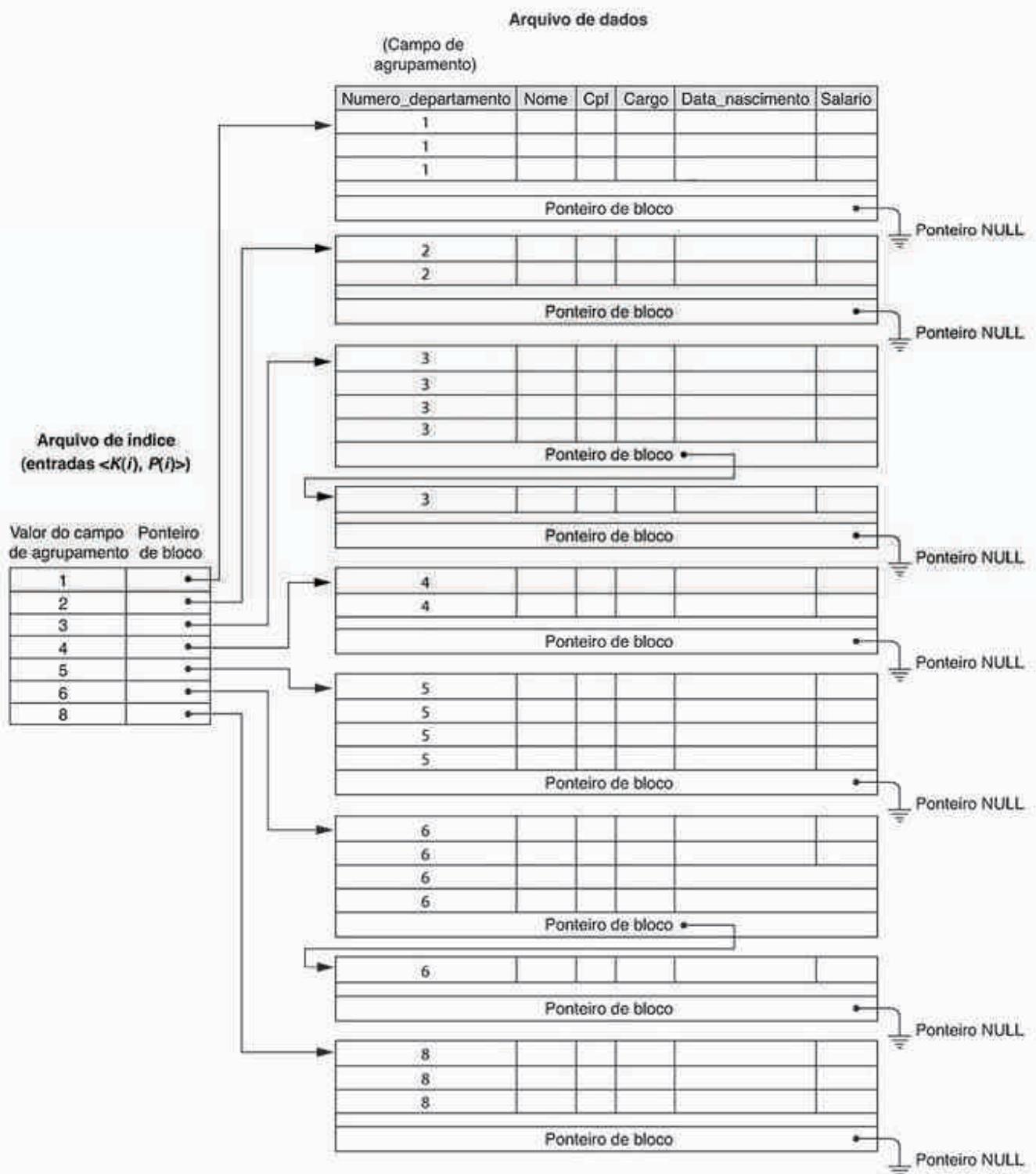


Figura 18.3

O índice de agrupamento com um cluster de bloco separado para cada grupo de registros que compartilham o mesmo valor para o campo de agrupamento.

dos, que contém o valor do campo para o registro e um ponteiro para o bloco em que o registro está armazenado ou para o próprio registro. Logo, tal índice é denso.

Mais uma vez, referimo-nos aos dois valores de campo da entrada de índice i como $\langle K(i), P(i) \rangle$. As entradas são ordenadas pelo valor de $K(i)$, de modo que podemos realizar uma pesquisa binária. Como os registros do arquivo de dados *não são* fisicamente ordenados pelos valores do campo de chave secundária, *não podemos* usar âncoras de bloco. É por isso que uma entrada de índice é criada para cada regis-

tro no arquivo de dados, em vez de para cada bloco, como no caso de um índice primário. A Figura 18.4 ilustra um índice secundário em que os ponteiros $P(i)$ nas entradas de índice são *ponteiros de bloco*, e não ponteiros de registro. Quando o bloco de disco apropriado é transferido para um buffer da memória principal, uma pesquisa pelo registro desejado no bloco pode ser executada.

Um índice secundário em geral precisa de mais espaço de armazenamento e tempo de busca maior do que um índice primário, devido a seu maior número de entradas. Porém, a *melhoria* no tempo de

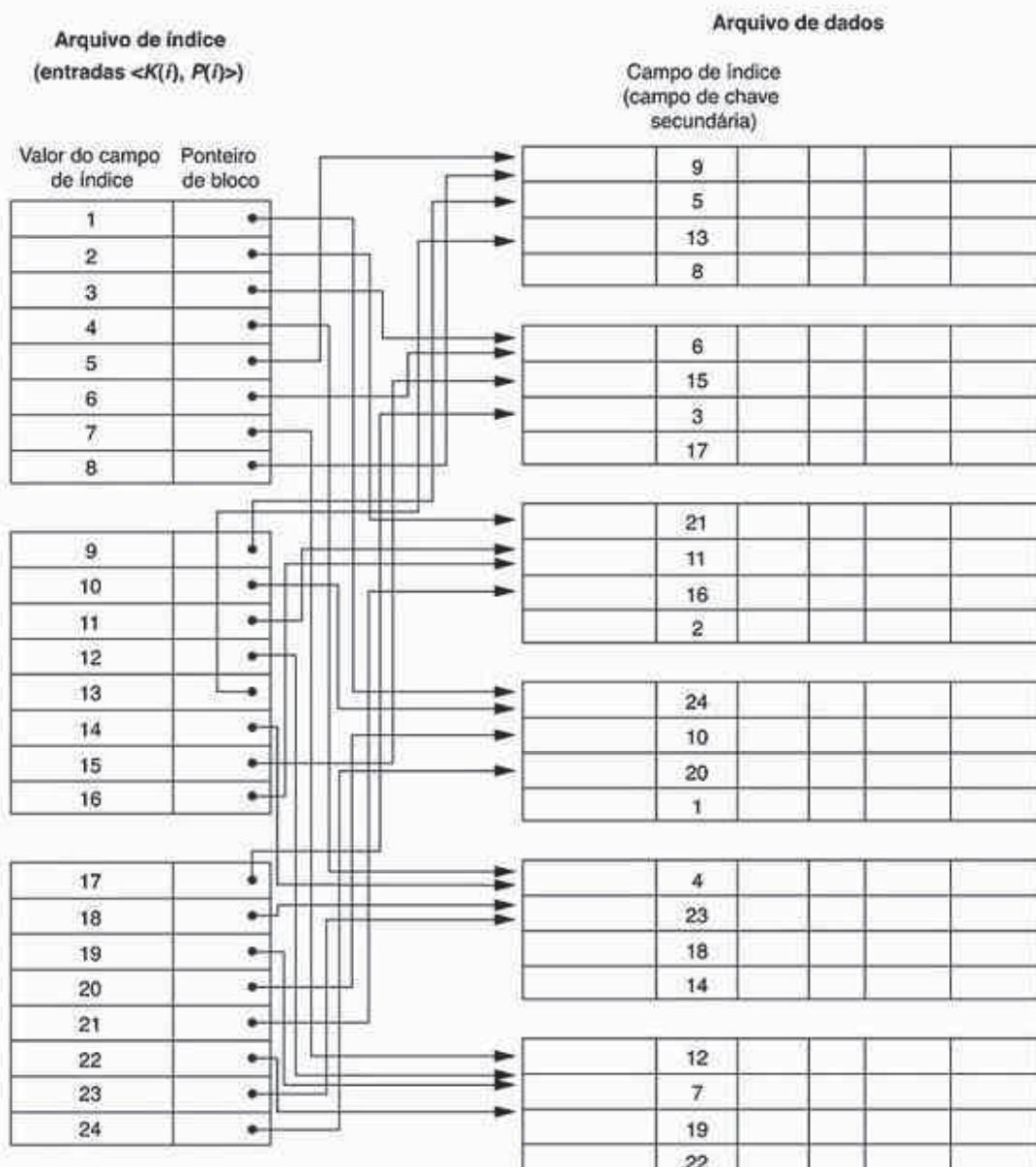


Figura 18.4

Um índice secundário denso (com ponteiros de bloco) em um campo de chave não ordenado de um arquivo.

pesquisa para um registro qualquer é muito maior para um índice secundário do que para um índice primário, visto que teríamos de fazer uma *pesquisa linear* no arquivo de dados se o índice secundário não existisse. Para um índice primário, poderíamos ainda usar uma pesquisa binária no arquivo principal, mesmo que o índice não existisse. O Exemplo 2 ilustra a melhoria no número de blocos acessados.

Exemplo 2. Considere o arquivo do Exemplo 1 com $r = 30.000$ registros de tamanho fixo de tamanho $R = 100$ bytes armazenados em um disco com tamanho de bloco $B = 1.024$ bytes. O arquivo tem $b = 3.000$ blocos, conforme calculado no Exemplo 1. Suponha que queiramos procurar um registro com um valor específico para a chave secundária — um campo de chave não ordenado do arquivo que tem $V = 9$ bytes de extensão. Sem o índice secundário, para fazer uma pesquisa linear no arquivo, seriam necessários $b/2 = 3.000/2 = 1.500$ acessos de bloco na média. Suponha que construíssemos um índice secundário nesse campo de chave não ordenado do arquivo. Como no Exemplo 1, um ponteiro de bloco tem $P = 6$ bytes de extensão, de modo que cada entrada de índice tem $R_i = (9 + 6) = 15$ bytes, e o fator de bloco para o índice é de $bfr_i = \lceil (B/R_i) \rceil = \lceil (1.024/15) \rceil = 68$ entradas por bloco. Em um índice secundário denso como este, o número total de entradas de índice r_i é igual ao número de registros no arquivo de dados, que é 30.000. O número de blocos necessários para o índice é, portanto, $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (3.000/68) \rceil = 442$ blocos.

Uma pesquisa binária nesse índice secundário precisa de $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 442) \rceil = 9$ acessos de bloco. Para procurar um registro usando o índice, precisamos de um acesso de bloco adicional ao arquivo de dados para um total de $9 + 1 = 10$ acessos de bloco — uma grande melhoria em relação aos 1.500 acessos de bloco necessários na média para uma pesquisa linear, mas ligeiramente pior que os 7 acessos de bloco exigidos para o índice primário. Essa diferença surgiu porque o índice primário era não denso e, portanto, menor, com apenas 45 blocos de extensão.

Também podemos criar um índice secundário em um campo *não chave, não ordenado* de um arquivo. Nesse caso, diversos registros no arquivo de dados podem ter o mesmo valor para o campo de índice. Existem várias opções para implementar tal índice:

- A opção 1 é incluir entradas de índice duplicadas com o mesmo valor $K(i)$ — um para cada registro. Este seria um índice denso.
- A opção 2 é ter registros de tamanho variável para as entradas de índice, com um campo repetitivo para o ponteiro. Mantemos uma lista de ponteiros $\langle P(i, 1), \dots, P(i, k) \rangle$ na entrada

de índice para $K(i)$ — um ponteiro para cada bloco que contém um registro cujo valor do campo de índice é igual a $K(i)$. Na opção 1 ou na opção 2, o algoritmo de pesquisa binária no índice deve ser modificado apropriadamente para considerar um número variável de entradas de índice por valor de chave de índice.

- A opção 3, a mais usada, é manter as próprias entradas de índice em um tamanho fixo e ter uma única entrada para cada *valor de campo de índice*, mas criar *um nível de indireção extra* para lidar com os múltiplos ponteiros. Nesse esquema não denso, o ponteiro $P(i)$ na entrada de índice $\langle K(i), P(i) \rangle$ aponta para um bloco de disco, que contém um *conjunto de ponteiros de registro*. Cada ponteiro de registro nesse bloco de disco aponta para um dos registros de arquivo de dados com valor $K(i)$ para o campo de índice. Se algum valor $K(i)$ ocorrer em muitos registros, de modo que seus ponteiros de registro não possam caber em um único bloco de disco, um cluster ou lista ligada de blocos é utilizada. Essa técnica é ilustrada na Figura 18.5. A recuperação por meio do índice requer um ou mais acessos de bloco adicionais, devido ao nível extra, mas os algoritmos para pesquisar o índice e (mais importante) para inserir novos registros no arquivo de dados são simples. Além disso, recuperações em condições de seleção complexas podem ser tratadas referindo-se aos ponteiros de registro, sem ter de recuperar muitos registros desnecessários do arquivo de dados (ver Exercício 18.23).

Observe que um índice secundário oferece uma ordenação lógica nos registros pelo campo de índice. Se acessarmos os registros na ordem das entradas no índice secundário, nós os obteremos na ordem do campo de índice. Os índices primário e de agrupamento assumem que o campo utilizado para a ordenação física dos registros no arquivo é o mesmo que o campo de índice.

18.1.4 Resumo

Para concluir esta seção, resumimos a discussão dos tipos de índice em duas tabelas. A Tabela 18.1 mostra as características do campo de índice de cada tipo de índice ordenado de único nível discutido — primário, de agrupamento e secundário. A Tabela 18.2 resume as propriedades de cada tipo de índice ao comparar o número de entradas de índice e especificar quais índices são densos e quais usam âncoras de bloco do arquivo de dados.

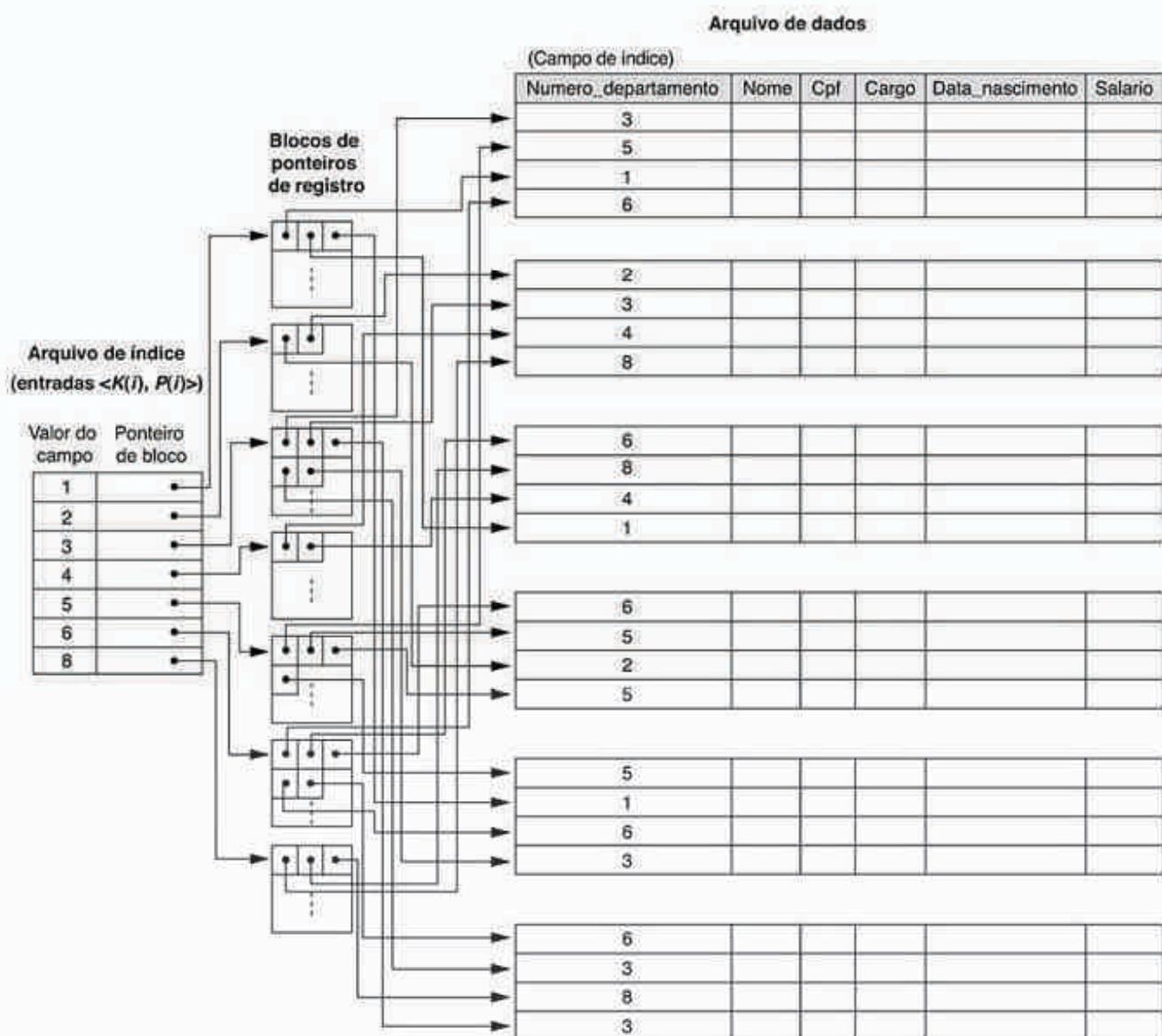


Figura 18.5

Um índice secundário (com ponteiros de registro) em um campo não chave implementado usando um nível de indireção, de modo que as entradas de índice são de tamanho fixo e têm valores de campo únicos.

Tabela 18.1

Tipos de índices baseados nas propriedades do campo de índice.

	Campo de índice usado para ordenação física do arquivo	Campo de índice não usado para ordenação física do arquivo
Campo de índice é chave	Índice primário	Índice secundário (chave)
Campo de índice é não chave	Índice de agrupamento	Índice secundário (não chave)

Tabela 18.2

Propriedades dos tipos de índice:

Tipo de índice	Número de entradas de índice (primeiro nível)	Denso ou não denso (esparso)	Ancoragem de bloco no arquivo de dados
Primário	Número de blocos no arquivo de dados	Não denso	Sim
Agrupamento	Número de valores de campo de índice distintos	Não denso	Sim/não ^a
Secundário (chave)	Número de registros no arquivo de dados	Denso	Não
Secundário (não chave)	Número de registros ^b ou número de valores de campo de índice distintos ^c	Denso ou não denso	Não

^a Sim, se cada valor distinto do campo de ordenação iniciar um novo bloco; caso contrário, não.^b Para a opção 1.^c Para as opções 2 e 3.

18.2 Índices multiníveis

Os esquemas de indexação que descrevemos até aqui envolvem um arquivo de índice ordenado. Uma pesquisa binária é aplicada ao índice para localizar ponteiros para um bloco de disco ou para um registro (ou registros) no arquivo que tem um valor específico de campo de índice. Uma pesquisa binária requer aproximadamente $(\log_2 b)$ acessos de bloco para um índice com b blocos, porque cada etapa do algoritmo reduz a parte do arquivo de índice que continuamos a pesquisar por um fator de 2. É por isso que recuperamos a função log à base 2. A ideia por trás de um índice multinível é reduzir a parte do índice que continuamos a pesquisar por bfr_1 , o fator de bloco para o índice, que é maior que 2. Logo, o espaço de pesquisa é reduzido muito mais rapidamente. O valor bfr_1 é chamado de fan-out do índice multinível, e vamos nos referir a ele com o símbolo fo . Enquanto dividimos o espaço de pesquisa de registro em duas metades a cada passo durante uma pesquisa binária, nós o dividimos n vezes (onde $n =$ o fan-out) a cada passo de pesquisa, usando o índice multinível. A pesquisa em um índice multinível requer aproximadamente $(\log_{fo} b)$ acessos de bloco, que é um número substancialmente menor do que para uma pesquisa binária se o fan-out for maior que 2. Na maioria dos casos, o fan-out é muito maior que 2.

Um índice multinível considera o arquivo de índice, ao qual nos referimos agora como o primeiro nível (ou de base) de um índice multinível, como um *arquivo ordenado* com um *valor distinto* para cada $K(i)$. Portanto, considerando o arquivo de índice de primeiro nível como um arquivo de dados classifi-

cado, podemos criar um índice primário para o primeiro nível; esse índice para o primeiro nível é chamado de segundo nível do índice multinível. Como o segundo nível é um índice primário, podemos usar âncoras de bloco de modo que o segundo nível tenha uma entrada para *cada bloco* do primeiro nível. O fator de bloco bfr_1 para o segundo nível — e para todos os níveis subsequentes — é o mesmo daquele para o índice do primeiro nível porque todas as entradas de índice são do mesmo tamanho; cada uma tem um valor de campo e um endereço de bloco. Se o primeiro nível tiver r_1 entradas, e o fator de bloco — que também é o fan-out — para o índice for $bfr_1 = fo$, então o primeiro nível precisará de $\lceil(r_1/fo)\rceil$ blocos, que é, portanto, o número de entradas r_2 necessárias no segundo nível do índice.

Podemos repetir esse processo para o segundo nível. O terceiro nível, que é um índice primário para o segundo nível, tem uma entrada para cada bloco do segundo nível, de modo que o número de entradas do terceiro nível é $r_3 = \lceil(r_2/fo)\rceil$. Observe que só exigimos um segundo nível se o primeiro nível precisar de mais de um bloco de armazenamento de disco e, de modo semelhante, exigimos um terceiro nível somente se o segundo nível precisar de mais de um bloco. Podemos repetir o processo anterior até que todas as entradas de algum nível de índice t caibam em um único bloco. Esse bloco no t -ésimo nível é chamado de nível de índice do topo.⁴ Cada nível reduz o número de entradas nos níveis anteriores por um fator de fo — o fan-out do índice —, de modo que podemos usar a fórmula $1 \leq (r_1/((fo)^t))$ para calcular t . Portanto, um índice multinível com r_1 entradas de primeiro nível terá aproximadamente t níveis, onde $t = \lceil(\log_{fo}(r_1))\rceil$.

^a O esquema de numeração para os níveis de índice usados aqui é o contrário do modo como os níveis normalmente são definidos para estruturas de dados de árvore. Nas estruturas de dados de árvore, t é referenciado como o nível 0 (zero), $t - 1$ é o nível 1, e assim por diante.

Ao pesquisar o índice, um único bloco de disco é recuperado em cada nível. Logo, t blocos de disco são acessados para uma pesquisa de índice, onde t é o número de níveis de índice.

O esquema multinível descrito aqui pode ser usado em qualquer tipo de índice — seja ele primário, de agrupamento ou secundário —, desde que o índice de primeiro nível tenha *valores distintos para $K(i)$ e entradas de tamanho fixo*. A Figura 18.6 mostra um índice multinível construído em um índice primário. O Exemplo 3 ilustra a melhoria no número de blocos acessados quando um índice multinível é utilizado para procurar um registro.

Exemplo 3. Suponha que o índice secundário denso do Exemplo 2 seja convertido em um índice multinível. Calculamos o fator de bloco de índice $b_{fr} = 68$ entradas de índice por bloco, que é também o fan-out fo para o índice multinível; o número de blocos de primeiro nível $b_1 = 442$ também foi calculado. O número de blocos de segundo nível será $b_2 = \lceil (b_1/fo) \rceil = \lceil (442/68) \rceil = 7$ blocos, e o número de blocos de terceiro nível será $b_3 = \lceil (b_2/fo) \rceil = \lceil (7/68) \rceil = 1$ bloco. Logo, o terceiro nível é o nível topo do índice, e $t = 3$. Para acessar um registro ao pesquisar um índice multinível, temos de acessar um bloco em cada nível mais um bloco do arquivo de dados, de modo que precisamos de $t + 1 = 3 + 1 = 4$ acessos de bloco. Compare isso com o Exemplo 2, onde foram necessários 10 acessos de bloco quando um índice de único nível e a pesquisa binária foram utilizados.

Observe que também poderíamos ter um índice primário multinível, que seria não denso. O Exercício 18.18(c) ilustra esse caso, em que temos de acessar o bloco de dados do arquivo antes de podermos determinar se o registro sendo pesquisado está no arquivo. Para um índice denso, isso pode ser determinado acessando o primeiro nível de índice (sem ter de acessar um bloco de dados), pois existe uma entrada de índice para cada registro no arquivo.

Uma organização de arquivo comum usada no processamento de dados comercial é um arquivo ordenado com um índice primário multinível em seu campo de chave de ordenação. Essa organização é chamada de arquivo sequencial indexado, e foi empregada em muitos dos primeiros sistemas IBM. A organização ISAM da IBM incorpora um índice de dois níveis que está relacionado de perto com a organização do disco em relação a cilindros e trilhas (ver Seção 17.2.1). O primeiro nível é um índice de cilindro, que tem o valor de chave de um registro de âncora para cada cilindro de um disk pack ocupado pelo arquivo e um ponteiro para o índice de trilha para o cilindro. O índice de tri-

lha tem o valor de chave de um registro de âncora para cada trilha no cilindro e um ponteiro para a trilha. Esta trilha pode então ser pesquisada de forma sequencial para o registro ou bloco desejado. A inserção é tratada por alguma forma de arquivo de overflow, que é mesclado periodicamente com o arquivo de dados. O índice é recriado durante a reorganização do arquivo.

O Algoritmo 18.1 esboça o procedimento de pesquisa para um registro em um arquivo de dados que utiliza um índice primário multinível não denso com t níveis. Referimo-nos à entrada i no nível j do índice como $\langle K_i(i), P_i(i) \rangle$ e procuramos um registro cujo valor de chave primária seja K . Consideraremos que quaisquer registros de overflow são ignorados. Se o registro estiver no arquivo, deverá haver alguma entrada no nível 1 com $K_1(i) \leq K < K_1(i+1)$ e o registro estará no bloco do arquivo de dados cujo endereço é $P_1(i)$. O Exercício 18.23 discute a modificação do algoritmo de pesquisa para outros tipos de índices.

Algoritmo 18.1. Pesquisando um índice primário multinível não denso com t níveis

(* Consideramos que a entrada de índice seja uma âncora de bloco que é a primeira chave por bloco. *)

$p \leftarrow$ endereço do bloco do nível topo do índice;
para $j \leftarrow t$ step -1 até 1 faça

início

lê o bloco de índice (no nível de índice j) cujo endereço é p ;

pesquisa bloco p para entrada i tal que $K_j(i) \leq K < K_j(i+1)$

(* se $K_j(i)$

é a última entrada no bloco, é suficiente satisfazer $K_j(i) \leq K$ *);

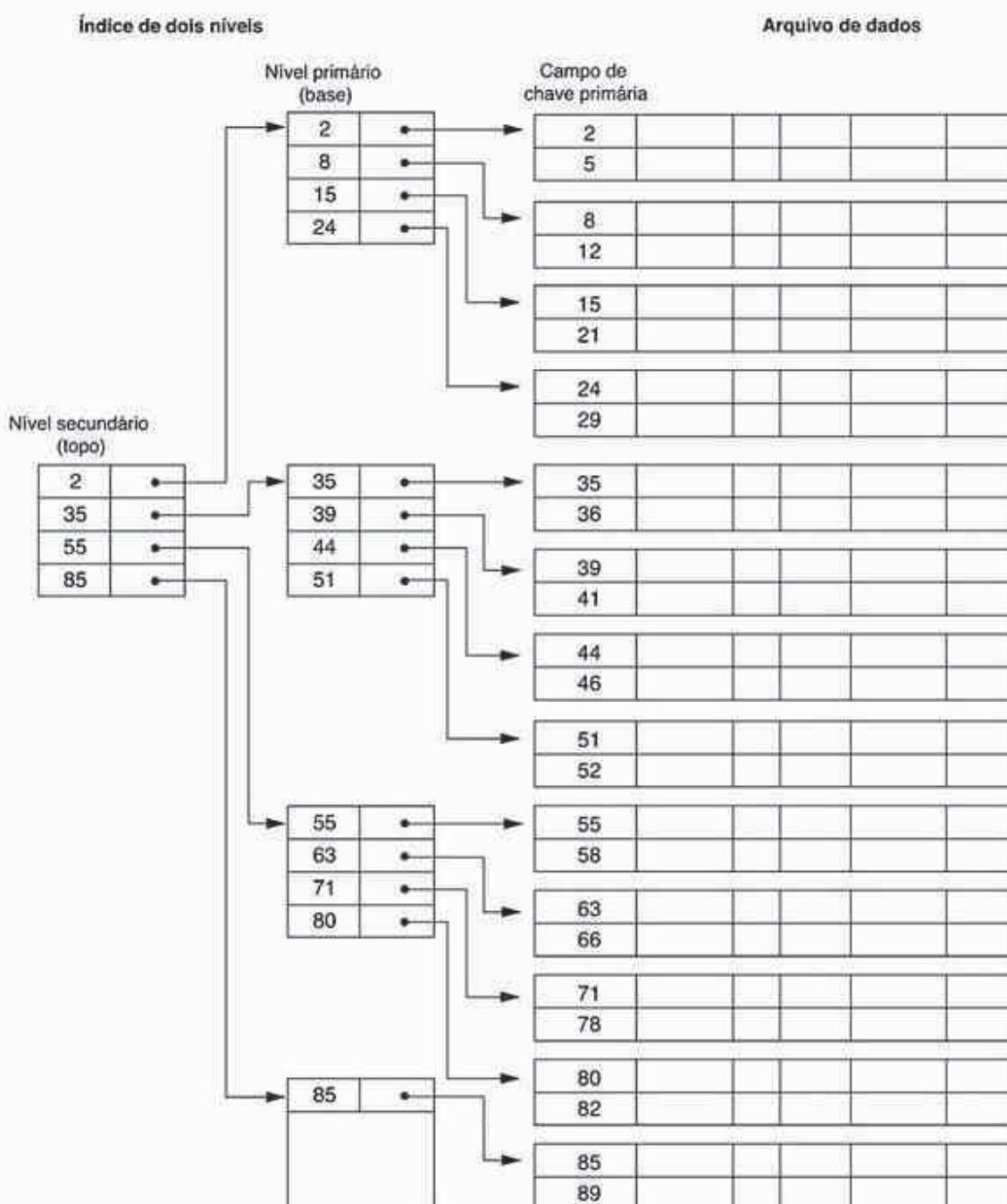
$p \leftarrow P_j(i)$ (* recupera ponteiro apropriado no nível de índice j *)

fim;

lê o bloco do arquivo de dados cujo endereço é p ;

pesquisa bloco p pelo registro com chave = K ;

Como vimos, um índice multinível reduz o número de blocos acessados quando se pesquisa um registro, dado seu valor de campo de indexação. Ainda enfrentamos os problemas de lidar com inserções e exclusões de índice, pois todos os níveis de índice são *arquivos ordenados fisicamente*. Para reter os benefícios do uso da indexação multinível enquanto reduzimos os problemas de inserção e exclusão de índice, os projetistas adotaram um índice multinível

**Figura 18.6**

Um índice primário de dois níveis semelhante à organização ISAM (*Indexed Sequential Access Method*).

chamado **índice multinível dinâmico**, que deixa algum espaço em cada um de seus blocos para inserir novas entradas e usa algoritmos apropriados de inserção/exclusão para criar e excluir novos blocos de índice quando o arquivo de dados cresce e encolhe. Ele normalmente é implementado ao usar estruturas de dados chamadas B-trees e B⁺-trees, que descreveremos na próxima seção.

18.3 Índices multiníveis dinâmicos usando B-trees e B⁺-trees

B-trees e B⁺-trees são casos especiais da famosa estrutura de dados de pesquisa, conhecida como árvore. Apresentamos rapidamente a terminologia usada na discussão de estruturas de dados de árvore. Uma árvore é formada de nós. Cada nó na árvore,

exceto pelo nó especial chamado **raiz**, tem um nó pai e zero ou mais nós filhos. O nó raiz não tem pai. Um nó que não tem filho algum é denominado **nó folha**; um nó não folha é chamado de **nó interno**. O nível de um nó é sempre um a mais que o nível de seu pai, com o nível do nó raiz sendo *zero*.⁵ Uma subárvore de um nó consiste nesse nó e todos os seus nós descendentes — seus nós filhos, os nós filhos de seus nós filhos, e assim por diante. Uma definição recursiva exata de uma subárvore é que ela consiste em um nó *n* e as subárvores de todos os nós filhos de *n*. A Figura 18.7 ilustra uma estrutura de dados em árvore. Nessa figura, o nó raiz é A, e seus nós filhos são B, C e D. Os nós E, J, C, G, H e K são nós folha. Como os nós folha estão em diferentes níveis da árvore, essa árvore é chamada de desbalanceada.

Na Seção 18.3.1, apresentamos árvores de pesquisa e depois discutimos sobre B-trees, que podem ser usadas como índices multiníveis dinâmicos para orientar a busca por registros em um arquivo de dados. Nós de B-tree são mantidos entre 50 e 100 por cento cheios, e os ponteiros para blocos de dados são armazenados nos nós internos e nós folha da estrutura da B-tree. Na Seção 18.3.2, abordamos as B⁺-trees, uma variação das B-trees em que os ponteiros para os blocos de dados de um arquivo são armazenados apenas em nós folha, que podem levar a menos níveis e índices de maior capacidade. Nos SGBDs prevalentes no mercado hoje em dia, a estrutura comum usada para indexação é B⁺-trees.

18.3.1 Árvores de pesquisa e B-trees

Uma árvore de pesquisa é um tipo especial de árvore utilizada para orientar a pesquisa por um registro, dado o valor de um dos campos do registro. Os índices multiníveis discutidos na Seção 18.2 podem ser imaginados como uma variação de uma árvore de pesquisa; cada nó no índice multinível pode ter inúmeros ponteiros *fo* e valores de chave *fo*, onde *fo* é o fan-out do índice. Os valores de campo de índice em cada nó nos guiam para o próximo nó, até que alcancemos o bloco do arquivo de dados que contém os registros solicitados. Ao seguir um ponteiro, restrinjimos nossa pesquisa em cada nível a uma subárvore da árvore de pesquisa e ignoramos todos os nós fora dessa subárvore.

Árvores de pesquisa. Uma árvore de pesquisa é ligeiramente diferente de um índice multinível. Uma árvore de pesquisa de ordem *p* é uma árvore tal que cada nó contém *no máximo p - 1* valores de pesquisa e *p* ponteiros na ordem $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$, onde $q \leq p$. Cada *P_i* é um ponteiro para um nó filho (ou um ponteiro NULL), e cada *K_i* é um valor de pesquisa de algum conjunto ordenado de valores. Todos os valores de pesquisa são considerados únicos.⁶ A Figura 18.8 ilustra um nó em uma árvore de pesquisa. Duas restrições precisam ser mantidas o tempo todo na árvore de pesquisa:

1. Em cada nó, $K_1 < K_2 < \dots < K_{q-1}$.

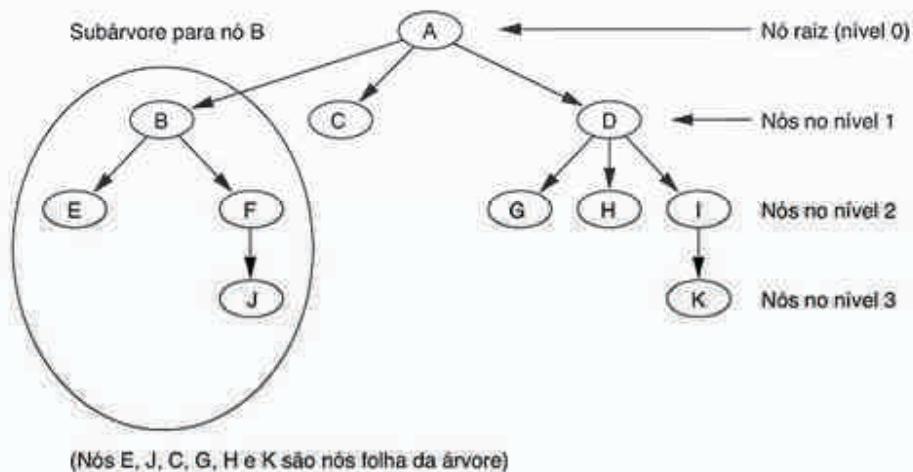


Figura 18.7

Uma estrutura de dados em árvore que mostra uma árvore desbalanceada.

⁵ Essa definição-padrão do nível de um nó de árvore, que usamos ao longo da Seção 18.3, é diferente daquela que demos para índices multiníveis na Seção 18.2.

⁶ Essa restrição pode ser relaxada. Se o índice for em um campo não chave, pode haver valores de pesquisa duplicados, e a estrutura do nó e as regras de navegação para a árvore podem ser modificadas.

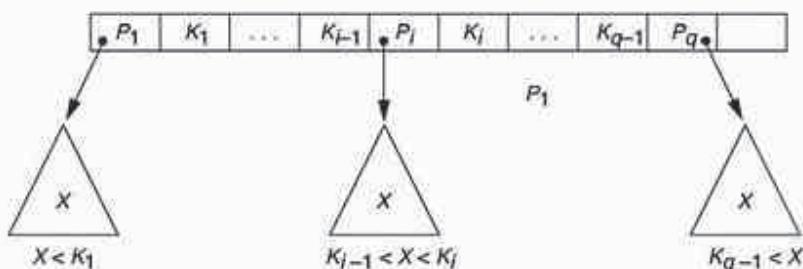


Figura 18.8

Um nó em uma árvore de pesquisa com ponteiros para subárvores abaixo dela.

- Para todos os valores X na subárvore apontada por P_i , temos $K_{i-1} < X < K_i$ para $1 < i < q$; $X < K_1$ para $i = 1$; e $K_{i-1} < X$ para $i = q$ (ver Figura 18.8).

Sempre que procuramos um valor X , seguimos o ponteiro P_i apropriado, de acordo com as fórmulas na condição 2. A Figura 18.9 ilustra uma árvore de pesquisa de ordem $p = 3$ e valores de pesquisa inteiros. Observe que alguns dos ponteiros P_i em um nó podem ser ponteiros NULL.

Podemos usar uma árvore de pesquisa como um mecanismo para procurar registros armazenados em um arquivo de disco. Os valores na árvore podem ser os valores de um dos campos do arquivo, chamado campo de pesquisa (que é o mesmo que o campo de índice se um índice multinível guiar a pesquisa). Cada valor de chave na árvore é associado a um ponteiro para o registro no arquivo de dados que tem esse valor. Como alternativa, o ponteiro poderia ser para o bloco de disco contendo esse registro. A própria árvore de pesquisa pode ser armazenada no disco ao atribuir cada nó de árvore a um bloco de disco. Quando um novo registro é inserido no arquivo, temos de atualizar a árvore de pesquisa inserindo uma entrada na árvore que contém o valor do campo de pesquisa do novo registro e um ponteiro para o novo registro.

São necessários algoritmos para inserir e excluir valores de pesquisa na árvore de pesquisa enquanto se mantêm as duas restrições anteriores. Em geral, esses algoritmos não garantem que uma árvore de pesquisa seja **balanceada**, significando que todos os seus nós folha estão no mesmo nível.⁷ A árvore da Figura 18.7 não é balanceada porque tem nós folha nos níveis 1, 2 e 3. Os objetivos para balancear uma árvore de pesquisa são os seguintes:

- Garantir que os nós sejam distribuídos por igual, de modo que a profundidade da árvore seja minimizada para determinado conjunto de chaves e que a árvore não fique distorcida, com alguns nós em níveis muito profundos.
- Tornar a velocidade de pesquisa uniforme, de modo que o tempo médio para encontrar qualquer chave aleatória seja aproximadamente o mesmo.

Embora minimizar o número de níveis na árvore seja um objetivo, outro objetivo implícito é garantir que a árvore de índice não precise de muita reestruturação quando os registros são inseridos e excluídos do arquivo principal. Assim, queremos que os nós sejam os mais cheios possíveis e não queremos que quaisquer nós sejam vazios se houver

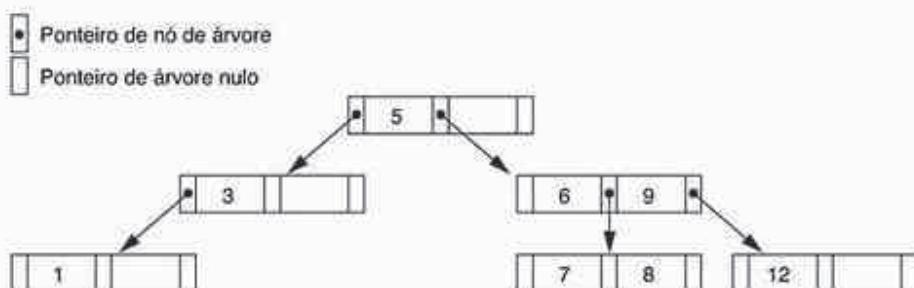


Figura 18.9

Uma árvore de pesquisa de ordem $p = 3$.

⁷ A definição de *balanceada* é diferente para árvores binárias. As árvores binárias平衡adas são conhecidas como *árvores AVL*.

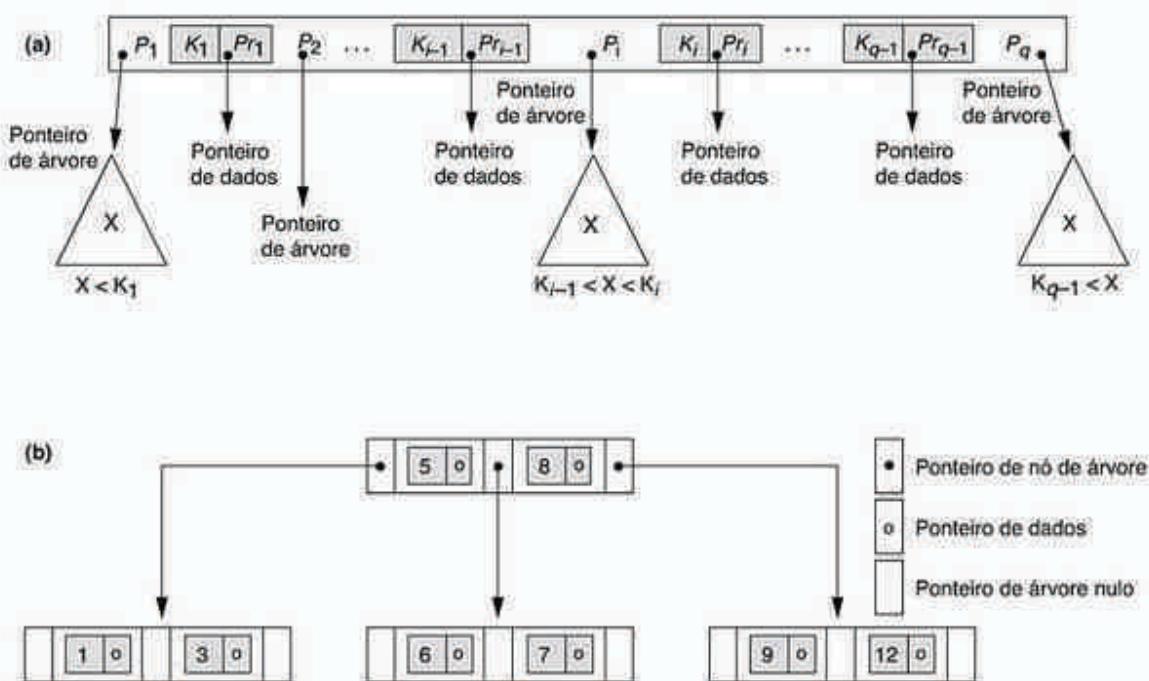


Figura 18.10

Estruturas B-tree. (a) Um nó em uma B-tree com $q - 1$ valores de pesquisa. (b) Uma B-tree de ordem $p = 3$. Os valores foram inseridos na ordem 8, 5, 1, 7, 3, 12, 9, 6.

muitas exclusões. A exclusão de registro pode deixar alguns nós na árvore quase vazios, desperdiçando assim o espaço de armazenamento e aumentando o número de níveis. A B-tree resolve esses dois problemas, especificando restrições adicionais na árvore de pesquisa.

B-trees. A B-tree tem restrições adicionais que garantem que a árvore sempre esteja balanceada e que o espaço desperdiçado pela exclusão, se houver, nunca se torne excessivo. Os algoritmos para inserção e exclusão, porém, tornam-se mais complexos a fim de manter essas restrições. Apesar disso, a maioria das inserções e exclusões são processos simples. Elas se tornam complicadas somente sob circunstâncias especiais — a saber, sempre que tentamos uma inserção em um nó que já está cheio ou uma exclusão de um nó que o torna cheio em menos da metade. De maneira mais formal, uma B-tree de ordem p , quando usada como uma estrutura de acesso em um campo de chave para pesquisar registros em um arquivo de dados, pode ser definida da seguinte forma:

1. Cada nó interno na B-tree (Figura 18.10(a)) tem a forma

$$\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$$

onde $q \leq p$. Cada P_i é um ponteiro de árvore — um ponteiro para outro nó na B-tree. Cada Pr_i é um ponteiro de dados⁸ — um ponteiro para o registro cujo valor do campo de chave de pesquisa é igual a K_i (ou ao bloco do arquivo de dados que contém esse registro).

2. Em cada nó, $K_1 < K_2 < \dots < K_{q-1}$.
 3. Para todos os valores de campo da chave de pesquisa X na subárvore apontada por P_i (a i -ésima subárvore; ver Figura 18.10(a)), temos:
- $K_{i-1} < X < K_i$ para $1 < i < q$; $X < K_i$ para $i = 1$; e $K_{q-1} < X$ para $i = q$.
4. Cada nó tem no máximo p ponteiros de árvore.
 5. Cada nó, exceto os nós raiz e folha, tem pelo menos $\lceil(p/2)\rceil$ ponteiros de árvore. O nó raiz tem pelo menos dois ponteiros de árvore, a menos que seja o único nó na árvore.

⁸Um ponteiro de dados é um endereço de bloco ou um endereço de registro; o último é basicamente um endereço de bloco e um deslocamento de registro dentro do bloco.

⁹Para detalhes sobre algoritmos de inserção e exclusão para B-trees, consulte Ramakrishnan e Gehrke (2003).

6. Um nó com q ponteiros de árvore, $q \leq p$, tem $q - 1$ valores de campo de chave de pesquisa (e, portanto, tem $q - 1$ ponteiros de dados).
7. Todos os nós folha estão no mesmo nível. Os nós folha têm a mesma estrutura dos nós internos, exceto que todos os seus *ponteiros de árvore* P_i são NULL.

A Figura 18.10(b) ilustra uma B-tree de ordem $p = 3$. Observe que todos os valores de pesquisa K na B-tree são únicos, pois consideramos que a árvore é usada como uma estrutura de acesso em um campo de chave. Se usarmos uma B-tree *em um campo não chave*, temos de mudar a definição dos ponteiros de arquivo P_i para apontar para um bloco — ou um cluster de blocos — que contenha os ponteiros para os registros de arquivo. Esse nível de indireção extra é semelhante à opção 3, discutida na Seção 18.1.3, para índices secundários.

Uma B-tree começa com um único nó raiz (que também é um nó folha) no nível 0 (zero). Quando o nó raiz está cheio com $p - 1$ valores de chave de pesquisa e tentamos inserir outra entrada na árvore, o nó raiz se divide em dois nós no nível 1. Somente o valor do meio é mantido no nó raiz, e o restante dos valores é dividido por igual entre os outros dois nós. Quando um nó não raiz está cheio e uma nova entrada é inserida nele, esse nó é dividido em dois nós no mesmo nível, e a entrada do meio é movida para o nó pai junto com dois ponteiros para os novos nós divididos. Se o nó pai estiver cheio, ele também é dividido. A divisão pode propagar por todo o caminho até o nó raiz, criando um novo nível se a raiz for dividida. Não vamos discutir os algoritmos para B-trees com detalhes neste livro,⁹ mas esboçamos os procedimentos de pesquisa e inserção para B⁺-trees na próxima seção.

Se a exclusão de um valor fizer que um nó fique com menos da metade cheio, ele é combinado com seus nós vizinhos, e isso também pode se propagar até a raiz. Logo, a exclusão pode reduzir o número de níveis da árvore. Foi demonstrado pelos analistas e pela simulação que, após diversas inserções e exclusões aleatórias em uma B-tree, os nós ficam aproximadamente 69 por cento cheios quando o número de valores na árvore se estabiliza. Isso também vale para B⁺-trees. Se isso acontecer, a divisão e a combinação de nós ocorrerão apenas raramente, de modo que a inserção e exclusão se tornam muito eficientes. Se o número de valores crescer, a árvore se expandirá sem problema — embora a divisão dos nós possa ocorrer, algumas inserções levarão mais tempo. Cada nó de B-tree pode ter *no máximo* p ponteiros de árvore, $p - 1$ ponteiros de dados, e $p - 1$ valores de campo de chave de pesquisa (ver Figura 18.10(a)).

Em geral, um nó de B-tree pode conter informações adicionais necessárias pelos algoritmos que manipulam a árvore, como o número de entradas q no nó e um ponteiro para o nó pai. A seguir, ilustramos como calcular o número de blocos e níveis para uma B-tree.

Exemplo 4. Suponha que o campo de pesquisa seja um campo de chave não ordenado, e construamos uma B-tree nesse campo com $p = 23$. Suponha que cada nó da B-tree esteja 69 por cento cheio. Cada nó, na média, terá $p * 0,69 = 23 * 0,69$ ou, aproximadamente, 16 ponteiros e, portanto, 15 valores de campo de chave de pesquisa. O fan-out médio $fo = 16$. Podemos começar na raiz e ver quantos valores e ponteiros podem existir, na média, em cada nível subsequente:

Raiz:	1 nó	15 entradas de chave	16 ponteiros
Nível 1:	16 nós	240 entradas de chave	256 ponteiros
Nível 2:	256 nós	3.840 entradas de chave	4.096 ponteiros
Nível 3:	4.096 nós	61.440 entradas de chave	

Em cada nível, calculamos o número de entradas de chave multiplicando o número total de ponteiros no nível anterior por 15, o número médio de entradas em cada nó. Assim, para determinado tamanho de bloco, tamanho de ponteiro e tamanho do campo de chave de pesquisa, uma B-tree de dois níveis mantém $3.840 + 240 + 15 = 4.095$ entradas na média; uma B-tree de três níveis mantém 65.535 entradas na média.

As B-trees às vezes são usadas como organizações de arquivo primárias. Nesse caso, *registros inteiros* são armazenados nos nós da B-tree, em vez de apenas as entradas <chave de pesquisa, ponteiro de registro>. Isso funciona bem para arquivos com um *número relativamente pequeno de registros* e um *tamanho de registro pequeno*. Caso contrário, o fan-out e o número de níveis tornam-se muito grande para permitir um acesso eficiente.

Resumindo, as B-trees oferecem uma estrutura de acesso multinível que é uma estrutura de árvore balanceada em que cada nó está cheio pelo menos até a metade. Cada nó em uma B-tree de ordem p pode ter no máximo $p - 1$ valores de pesquisa.

18.3.2 B⁺-trees

A maioria das implementações de um índice multínivel dinâmico utiliza uma variação da estrutura de dados da B-tree chamada B⁺-tree. Em uma B-tree, cada valor do campo de pesquisa aparece uma vez em algum nível na árvore, junto com um ponteiro de dados. Em uma B⁺-tree, os ponteiros de dados são armazenados *apenas nos nós folha* da árvore; logo, a estrutura

dos nós folha difere da estrutura dos nós internos. Os nós folha têm uma entrada para *cada* valor do campo de pesquisa, junto com um ponteiro de dados para o registro (ou para o bloco que contém esse registro), se o campo de pesquisa for um campo de chave. Para um campo de pesquisa não chave, o ponteiro aponta para um bloco que contém ponteiros para os registros do arquivo de dados, criando um nível de indireção extra.

Os nós folha da B*-tree normalmente são ligados para oferecer acesso ordenado no campo de pesquisa aos registros. Esses nós folha são semelhantes ao primeiro nível (base) de um índice. Os nós internos da B*-tree correspondem aos outros níveis de um índice multinível. Alguns valores de campo de pesquisa dos nós folha são *repetidos* nos nós internos da B*-tree para guiar a pesquisa. A estrutura dos *nós internos* de uma B*-tree de ordem p (Figura 18.11(a)) é a seguinte:

1. Cada nó interno tem a forma $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$
2. Em cada nó interno, $K_1 < K_2 < \dots < K_{q-1}$.
3. Para todos os valores de campo de pesquisa X na subárvore apontada por P_i , temos $K_{i-1} < X \leq K_i$, para $1 < i < q$; $X \leq K_1$ para $i = 1$; e $K_{q-1} < X$ para $i = q$ (ver Figura 18.11(a)).¹⁰
4. Cada nó interno tem, no máximo, p ponteiros de árvore.

5. Cada nó interno, exceto a raiz, tem pelo menos $\lceil(p/2)\rceil$ ponteiros de árvore. O nó raiz tem pelo menos dois ponteiros de árvore, se for um nó interno.
6. Um nó interno com q ponteiros, $q \leq p$, tem $q - 1$ valores de campo de pesquisa.

A estrutura dos *nós folha* de uma B*-tree de ordem p (Figura 18.11(b)) é a seguinte:

1. Cada nó folha tem a forma

$\langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_{\text{próximo}}$

onde $q \leq p$, cada Pr_i é um ponteiro de dados e $P_{\text{próximo}}$ aponta para o próximo *nó folha* da B*-tree.

2. Em cada nó folha, $K_1 \leq K_2 \dots \leq K_{q-1}$, $q \leq p$.
3. Cada Pr_i é um ponteiro de dados que aponta para o registro cujo valor do campo de pesquisa é K_i , ou para um bloco de arquivo que contém o registro (ou para um bloco de ponteiros de registro que aponta para registros cujo valor do campo de pesquisa é K_i , se o campo de pesquisa não for uma chave).
4. Cada nó folha tem pelo menos $\lceil(p/2)\rceil$ valores.
5. Todos os nós folha estão no mesmo nível.

Os ponteiros nos nós internos são *três ponteiros* para blocos que são nós de árvore, enquanto os ponteiros nos nós folha são *ponteiros de dados* para os registros ou blocos do arquivo de dados — exceto para o ponteiro $P_{\text{próximo}}$, que é um ponteiro de árvore para o próximo nó folha. Ao começar no nó folha mais à es-

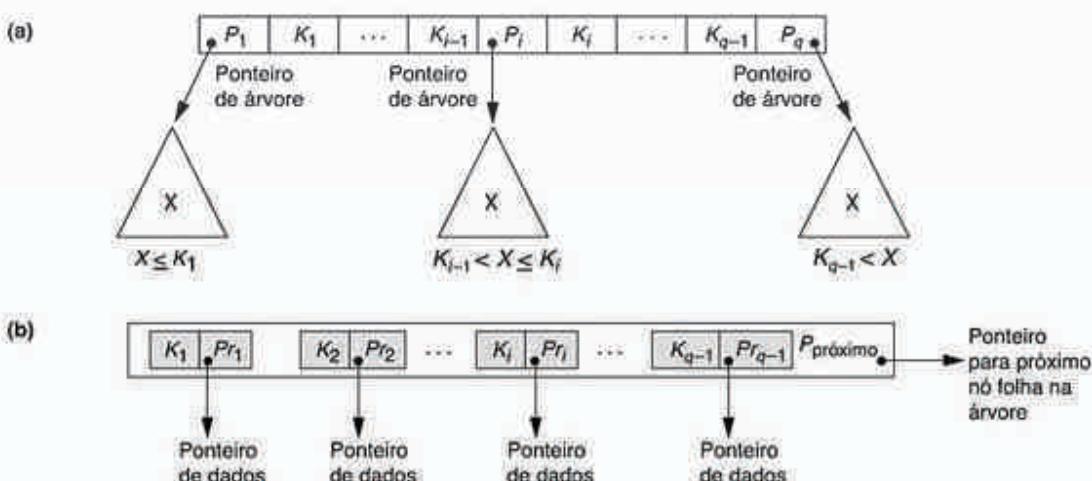


Figura 18.11

Os nós de uma B*-tree. (a) Nô interno de uma B*-tree com $q - 1$ valores de pesquisa. (b) Nô folha de uma B*-tree com $q - 1$ valores de pesquisa e $q - 1$ ponteiros de dados.

¹⁰ Nossa definição segue Knuth (1998). Pode-se definir uma B*-tree de forma diferente ao trocar os símbolos $<$ e \leq ($K_{i-1} \leq X < K_i$; $K_{q-1} \leq X$), mas os princípios continuam sendo os mesmos.

querda, é possível atravessar os nós folha como uma lista ligada, usando os ponteiros $P_{\text{próximo}}$. Isso oferece acesso ordenado aos registros de dados no campo de índice. Um ponteiro P_{anterior} também pode ser incluído. Para uma B⁺-tree em um campo não chave, é necessário um nível extra de indireção, semelhante ao que mostramos na Figura 18.5, de modo que os P_r ponteiros são ponteiros de bloco para os blocos que contêm um conjunto de ponteiros de registro para os registros reais no arquivo de dados, conforme discutimos na opção 3 da Seção 18.1.3.

Como as entradas nos *nós internos* de uma B⁺-tree incluem valores de pesquisa e ponteiros de árvore sem quaisquer ponteiros de dados, mas entradas podem ser compactadas em um nó interno de uma B⁺-tree do que para uma B-tree semelhante. Assim, para o mesmo tamanho de bloco (nó), a ordem p será maior para a B⁺-tree do que para a B-tree, conforme ilustramos no Exemplo 5. Isso pode levar a menos níveis de B⁺-tree, melhorando o tempo de pesquisa. Como as estruturas para nós internos e folha de uma B⁺-tree são diferentes, a ordem p pode ser diferente. Usaremos p para indicar a ordem para *nós internos* e p_{folha} para indicar a ordem para *nós folha*, que definimos como sendo o número máximo de ponteiros de dados em um nó folha.

Exemplo 5. Para calcular a ordem p de uma B⁺-tree, suponha que o campo de chave de pesquisa seja $V = 9$ bytes de extensão, o tamanho do bloco seja $B = 512$ bytes, um ponteiro de registro seja $Pr = 7$ bytes e um ponteiro de bloco seja $P = 6$ bytes. Um nó interno da B⁺-tree pode ter até p ponteiros de árvore e $p - 1$ valores de campo de pesquisa; estes precisam caber em um único bloco. Logo, temos:

$$\begin{aligned}(p * P) + ((p - 1) * V) &\leq B \\ (P * 6) + ((P - 1) * 9) &\leq 512 \\ (15 * p) &\leq 521\end{aligned}$$

Podemos escolher p para ser o maior valor que satisfaça a desigualdade acima, que gera $p = 34$. Isso é maior do que o valor de 23 para a B-tree (fica para o leitor a tarefa de calcular a ordem da B-tree ao considerar ponteiros do mesmo tamanho), resultando em um fan-out maior e mais entradas em cada nó interno de uma B⁺-tree do que na B-tree correspondente. Os nós folha da B⁺-tree terão o mesmo número de valores e ponteiros, exceto que os ponteiros são ponteiros de dados e um ponteiro de próximo. Logo, a ordem p_{folha} para os nós folha pode ser calculada da seguinte forma:

$$\begin{aligned}(p_{\text{folha}} * (Pr + V)) + P &\leq B \\ (p_{\text{folha}} * (7 + 9)) + 6 &\leq 512 \\ (16 * p_{\text{folha}}) &\leq 506\end{aligned}$$

Segue-se que cada nó folha pode manter até $p_{\text{folha}} = 31$ combinações de ponteiro de valor/dados de chave, considerando que os ponteiros de dados são ponteiros de registro.

Assim como a B-tree, podemos precisar de informações adicionais — para implementar os algoritmos de inserção e exclusão — em cada nó. Essa informação pode incluir o tipo de nó (interno ou folha), o número de entradas atuais q no nó e ponteiros para os nós pai e irmão. Logo, antes de fazermos os cálculos acima para p e p_{folha} , devemos reduzir o tamanho do bloco pela quantidade de espaço necessária para toda essa informação. O próximo exemplo ilustra como podemos calcular o número de entradas em uma B⁺-tree.

Exemplo 6. Suponha que criemos uma B⁺-tree no campo do Exemplo 5. Para calcular o número aproximado de entradas na B⁺-tree, consideramos que cada nó está 69 por cento cheio. Na média, cada nó interno terá $34 * 0,69$ ou, aproximadamente, 23 ponteiros e, portanto, 22 valores. Cada nó folha, em média, manterá $0,69 * p_{\text{folha}} = 0,69 * 31$ ou, aproximadamente, 21 ponteiros de registro de dados. Uma B⁺-tree terá o seguinte número médio de entradas em cada nível:

Paiz	1 nó	22 entradas de chave	23 ponteiros
Nível 1:	23 nós	506 entradas de chave	529 ponteiros
Nível 2:	529 nós	11.638 entradas de chave	12.167 ponteiros
Nível	12.167 nós	255.507 ponteiros de registro de dados	

Para o tamanho do bloco, tamanho do ponteiro e tamanho do campo de pesquisa dados acima, uma B⁺-tree de três níveis mantém até 255.507 ponteiros de registro, com a média de 69 por cento de ocupação de nós. Compare isso com as 65.535 entradas para a B-tree correspondente do Exemplo 4. Esse é o principal motivo para as B⁺-trees serem preferidas às B-trees como índices para arquivos de banco de dados.

Pesquisa, inserção e exclusão com B⁺-trees. O Algoritmo 18.2 esboça o procedimento usando a B⁺-tree como estrutura de acesso para pesquisar um registro. O Algoritmo 18.3 ilustra o procedimento para inserir um registro em um arquivo com uma estrutura de acesso de B⁺-tree. Esses algoritmos consideram a existência de um campo de pesquisa de chave, e eles devem ser modificados apropriadamente para o caso de uma B⁺-tree em um campo não chave. Ilustramos a inserção e a exclusão com um exemplo.

Algoritmo 18.2. Pesquisando um registro com o valor do campo de chave de pesquisa K , usando uma B⁺-tree

```

 $n \leftarrow$  bloco contendo nó raiz da B+-tree;
lê bloco  $n$ ;
enquanto ( $n$  não é um nó folha da B+-tree) do
    início
         $q \leftarrow$  número de ponteiros de árvore no nó  $n$ ;
        se  $K \leq n.K_i$  (* $n.K_i$  refere-se ao  $i$ -ésimo valor de
            campo de pesquisa no nó  $n$ )
            então  $n \leftarrow n.P_i$  (* $n.P_i$  refere-se ao  $i$ -ésimo
                ponteiro de árvore no nó  $n$ )
            se não if  $K > n.K_{q-1}$ 
                então  $n \leftarrow n.P_q$ 
                se não início
                    procura no nó  $n$  uma entrada  $i$  tal
                    que  $n.K_{i-1} < K \leq n.K_i$ ;
                     $n \leftarrow n.P_i$ 
                    fim;
                fim;
            lê bloco  $n$ 
            fim;
        procura no bloco  $n$  uma entrada  $(K_p, Pr)$  com  $K$ 
        =  $K_i$ ; (* procura nó folha *)
        se encontrado
            então lê bloco do arquivo de dados com en-
                dereço  $Pr$ , e recupera registro
            se não o registro com valor do campo de pes-
               quisa  $K$  não está no arquivo de dados;
```

Algoritmo 18.3. Inserindo um registro com valor do campo de chave de pesquisa K em uma B⁺-tree de ordem p

```

 $n \leftarrow$  bloco contendo nó raiz da B+-tree;
lê bloco  $n$ ; define pilha  $S$  como vazia;
enquanto ( $n$  não é nó folha da B+-tree) do
    início
        coloca endereço de  $n$  na pilha  $S$ ;
        (*pilha  $S$  mantém nós pai que são necessá-
            rios no caso de divisão*)
         $q \leftarrow$  número de ponteiros de árvore no nó  $n$ ;
        se  $K \leq n.K_i$  (* $n.K_i$  refere-se ao  $i$ -ésimo valor
            do campo de pesquisa no nó  $n$ )
            então  $n \leftarrow n.P_i$  (* $n.P_i$  refere-se ao  $i$ -ésimo
                ponteiro de árvore no nó  $n$ )
            se não if  $K > n.K_{q-1}$ 
                então  $n \leftarrow n.P_q$ 
                se não início
                    procura no nó  $n$  uma entrada  $i$  tal que
                     $n.K_{i-1} < K \leq n.K_i$ ;
                     $n \leftarrow n.P_i$ 
                    fim;
```

```

lê bloco  $n$ 
fim;
procura no bloco  $n$  uma entrada  $(K_p, Pr)$  com  $K = K_i$ ;
(*procura nó folha  $n$ )
se encontrado
    então registro já no arquivo; não pode inserir
    se não (*insere registro na B+-tree para apontar
        para registro*)
        início
            cria entrada  $(K, Pr)$  onde  $Pr$  aponta para o
            novo registro;
            se nó folha  $n$  não está cheio
                então insere entrada  $(K, Pr)$  na posição
                correta no nó folha  $n$ 
                se não início (*nó folha  $n$  está cheio com
                    ponteiros de registro  $p_{folha}$ ; é dividido*)
                    copia  $n$  para  $temp$  (* $temp$  é um nó folha
                    maior para manter entradas extras*);
                    insere entrada  $(K, Pr)$  em  $temp$  na posi-
                    ção correta;
                    (* $temp$  agora mantém  $p_{folha} + 1$  entradas
                     na forma  $(K_p, Pr)$ *)
                     $new \leftarrow$  um novo nó folha vazio para a
                     árvore;  $new.P_{próximo} \leftarrow n.P_{próximo}$ ;
                     $j \leftarrow \lceil(p_{folha} + 1)/2\rceil$ ;
                     $n \leftarrow$  primeiras  $j$  entradas em  $temp$  (até
                     entrada  $(K_p, Pr)$ );  $n.P_{próximo} \leftarrow new$ ;
                     $new \leftarrow$  entradas restantes em  $temp$ ;  $K$ 
                      $\leftarrow K_i$ ;
                    (*agora temos de mover  $(K, new)$  e inserir
                     no nó interno pai;
                     porém, se o pai estiver cheio, a divisão
                     pode se propagar*)
                    finished  $\leftarrow$  false;
                    repita
                    se pilha  $S$  está vazia
                        então (*nenhum nó pai; novo nó raiz
                            é criado para a árvore*)
                            início
                                 $root \leftarrow$  um novo nó interno vazio
                                para a árvore;
                                 $root \leftarrow \langle n, K, new \rangle$ ; finished  $\leftarrow$ 
                                true;
                                fim
                            se não início
                                 $n \leftarrow$  remove da pilha  $S$ ;
                                se nó interno  $n$  não está cheio
                                    então
                                        início (*nó pai não cheio;
                                            não divide*)
                                            insere  $(K, new)$  na posição
```

```

correta no nó interno  $n$ ;
finished  $\leftarrow$  true
fim
se não início (*nó interno  $n$  está cheio com ponteiros de árvore  $p$ ;  

    condição de overflow; nó é dividido*)
    copia  $n$  para  $temp$  (* $temp$  é um nó interno maior*);
    insere ( $K$ ,  $new$ ) em  $temp$  na posição correta;
    (* $temp$  agora tem  $p + 1$  ponteiros de árvore*)
     $new \leftarrow$  um novo nó interno vazio para a árvore;
     $j \leftarrow ((p + 1)/2)$ ;
     $n \leftarrow$  entradas até o ponteiro de árvore  $P_i$  em  $temp$ ;
    (* $n$  contém  $<P_1, K_1, P_2, K_2, \dots, P_{i-1}, K_{i-1}, P_i>$ )
     $new \leftarrow$  entradas até o ponteiro de árvore  $P_{i+1}$  em  $temp$ ;
    (* $new$  contém  $<P_{i+1}, K_{i+1}, \dots, K_{p-1}, P_p, K_p, P_{p+1}>$ )
     $K \leftarrow K_i$ 
    (*agora temos de mover ( $K$ ,  $new$ ) e inserir no nó interno pai*)
fim
fim
until terminado
fim;
fim;

```

A Figura 18.12 ilustra a inserção de registros em uma B'-tree de ordem $p = 3$ e $p_{folha} = 2$. Primeiro, observamos que a raiz é o único nó na árvore, de modo que também é um nó folha. Assim que mais de um nível é criado, a árvore é dividida em nós internos e nós folha. Observe que *cada valor de chave precisa existir no nível de folha*, pois todos os ponteiros de dados estão no nível de folha. Contudo, somente alguns valores existem nos nós internos para guiar a pesquisa. Observe também que cada valor que aparece em um nó interno também aparece como o *valor mais à direita* no nível de folha da subárvore apontada pelo ponteiro de árvore à esquerda do valor.

Quando um *nó folha* está cheio e uma nova entrada é inserida lá, ele *estoura* e precisa ser dividido. As primeiras entradas $j = \lfloor((p_{folha} + 1)/2)\rfloor$ no nó

original são mantidas lá, e as entradas restantes são movidas para um novo nó folha. O j -ésimo valor de pesquisa é replicado no nó interno pai, e um ponteiro extra para o novo nó é criado no pai. Estes precisam ser inseridos no nó pai em sua sequência correta. Se o nó interno pai estiver cheio, o novo valor fará que ele estoure também, de modo que precisará ser dividido. As entradas no nó interno até P_j — o j -ésimo ponteiro de árvore após inserir o novo valor e ponteiro, onde $j = \lfloor((p + 1)/2)\rfloor$ — são mantidas, enquanto o j -ésimo valor de pesquisa é movido para o pai, não replicado. Um novo nó interno manterá as entradas de P_{j+1} para o final das entradas no nó (ver Algoritmo 18.3). Essa divisão pode se propagar para cima até criar um novo nó raiz e, portanto, um novo nível para a B'-tree.

A Figura 18.13 ilustra a exclusão de uma B'-tree. Quando uma entrada é excluída, ela sempre é removida do nível folha. Se ocorrer em um nó interno, ela também precisa ser removida de lá. No último caso, o valor à sua esquerda no nó folha precisa substituí-lo no nó interno, pois esse valor agora é a entrada mais à direita na subárvore. A exclusão pode causar underflow, reduzindo o número de entradas no nó folha abaixo do mínimo exigido. Nesse caso, tentamos encontrar um nó folha irmão — um nó folha diretamente à esquerda ou à direita do nó com underflow — e redistribuir as entradas entre o nó e seu irmão, de modo que ambos estejam pelo menos cheios pela metade. Caso contrário, o nó é mesclado com seus irmãos e o número de nós folha é reduzido. Um método comum é tentar redistribuir entradas com o irmão da esquerda; se isso não for possível, é feita uma tentativa para redistribuir com o irmão da direita. Se isso também não for possível, os três nós são mesclados em dois nós folha. Nesse caso, o underflow pode se propagar para nós internos, porque são necessários menos ponteiros de árvore e valor de pesquisa. Isso pode propagar e reduzir os níveis da árvore.

Observe que a implementação dos algoritmos de inserção e exclusão pode exigir ponteiros pai e irmão para cada nó, ou o uso de uma pilha, como no Algoritmo 18.3. Cada nó também deve incluir o número de entradas nele e seu tipo (folha ou interno). Uma alternativa é implementar a inserção e a exclusão como procedimentos recursivos.¹¹

Variações das B-trees e B'-trees. Para concluir esta seção, mencionamos rapidamente algumas variações das B-trees e B'-trees. Em alguns casos, a restrição 5 na B-tree (ou para os nós internos da B'-tree, exceto o nó raiz), que exige que cada nó esteja cheio pelo menos pela metade, pode ser alterada para exigir que cada nó esteja pelo menos dois terços cheio. Nesse caso, a

¹¹ Para obter mais detalhes sobre algoritmos de inserção e exclusão para B'-trees, consulte Ramakrishnan e Gehrke (2003).

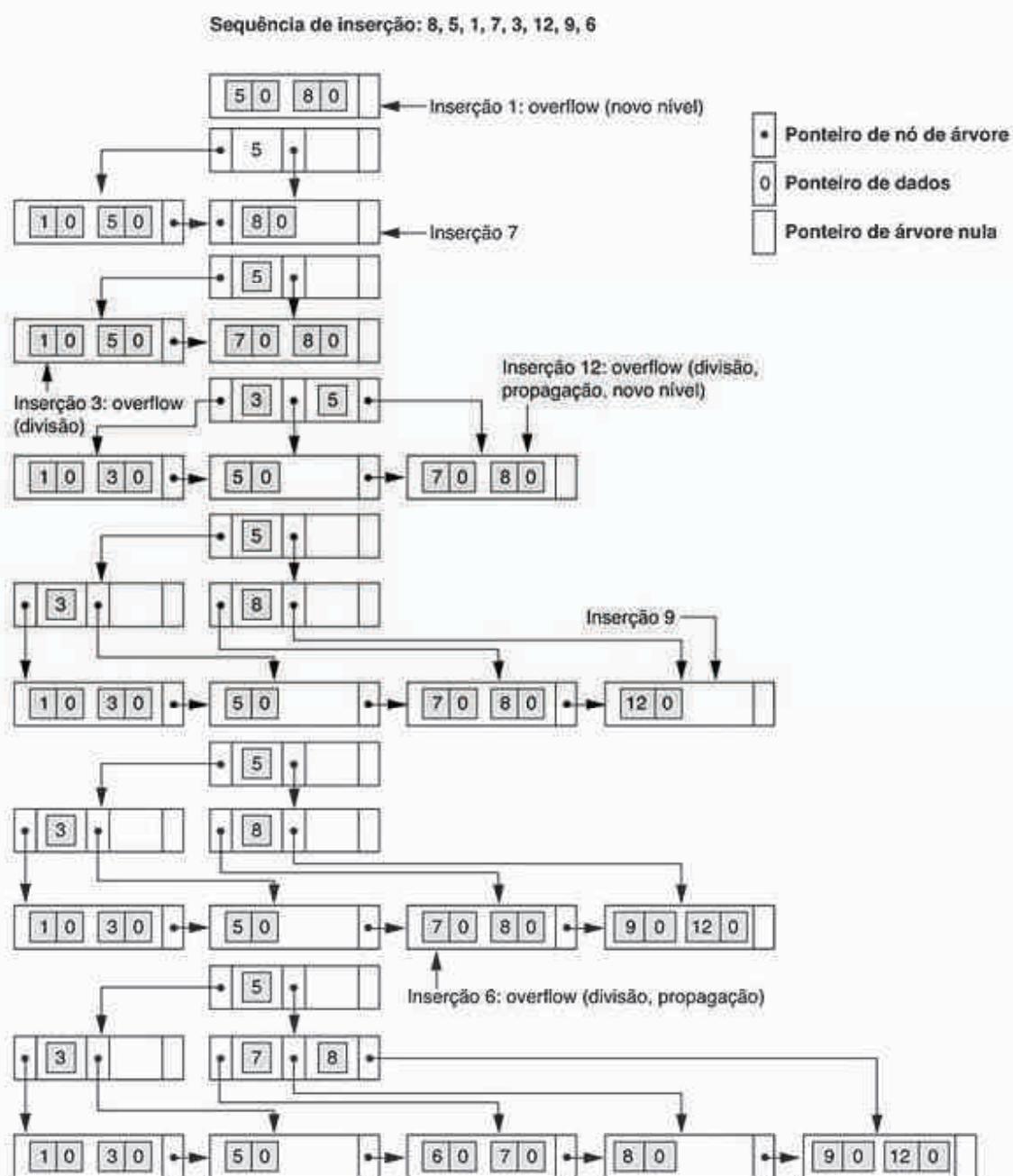


Figura 18.12

Exemplo de uma inserção em uma B*-tree com $p = 3$ e $p_{\min} = 2$.

B-tree é chamada de B*-tree. Em geral, alguns sistemas permitem que o usuário escolha um fator de preenchimento entre 0,5 e 1,0, no qual o último significa que os nós da B-tree (índice) devem estar completamente cheios. Também é possível especificar dois fatores de preenchimento para uma B*-tree: um para o nível folha e um para os nós internos da árvore. Quando o índice é construído inicialmente, cada nó é preenchido

até aproximadamente os fatores de preenchimento especificados. Alguns investigadores sugeriram relaxar o requisito de que um nó esteja cheio pela metade, e, em vez disso, permitir que um nó se torne completamente vazio antes da mesclagem, para simplificar o algoritmo de exclusão. Estudos de simulação mostram que isso não desperdiça muito espaço adicional sob inserções e exclusões distribuídas aleatoriamente.

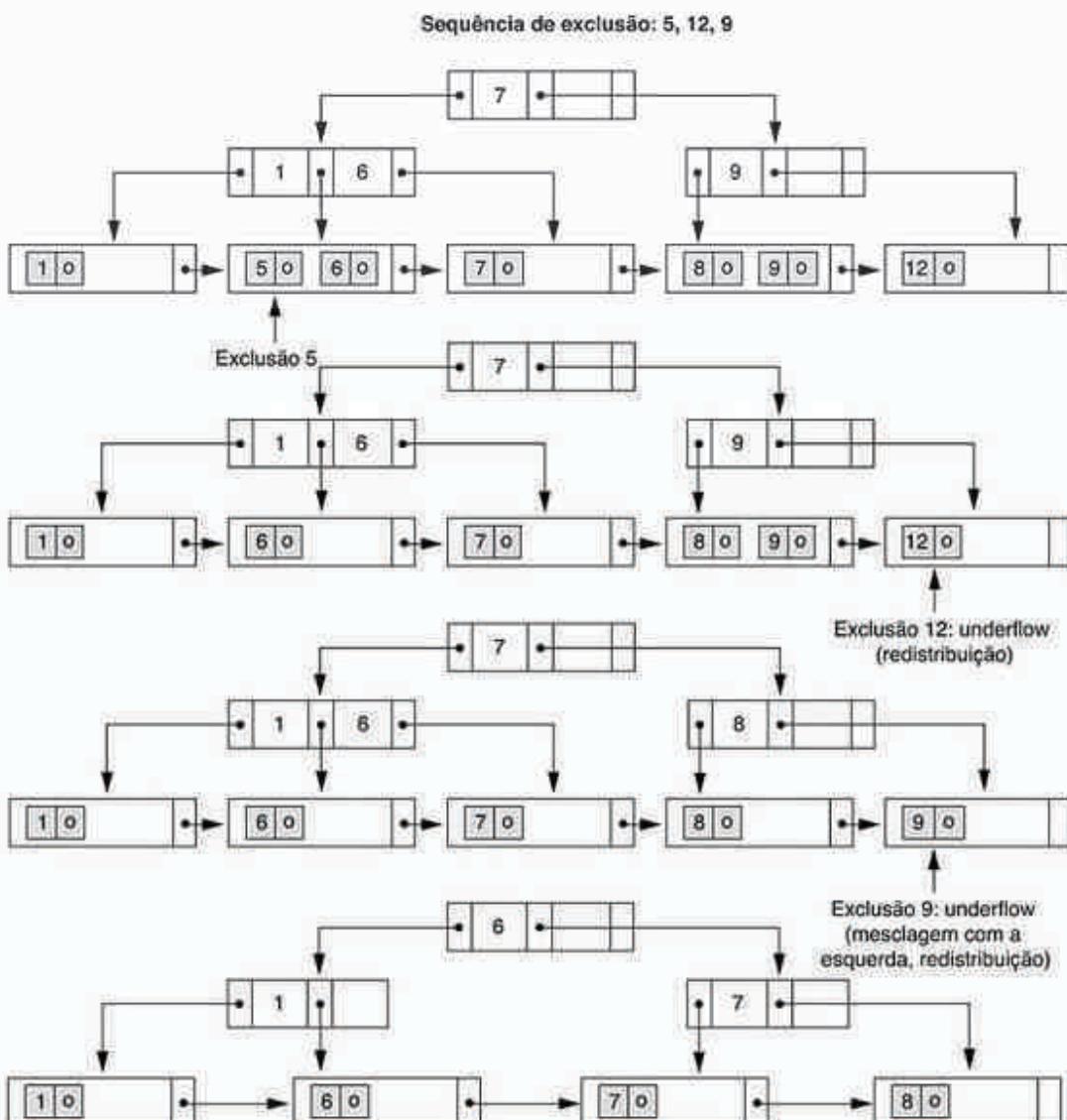


Figura 18.13

Um exemplo de exclusão de uma B⁺-tree.

18.4 Índices em múltiplas chaves

Em nossa discussão até aqui, consideramos que as chaves primária ou secundária nas quais os arquivos eram acessados eram atributos (campos) únicos. Em muitas solicitações de recuperação e atualização, vários atributos são envolvidos. Se certa combinação de atributos for usada com frequência, é vantajoso configurar uma estrutura de acesso para oferecer acesso eficaz por um valor de chave que é uma combinação desses atributos.

Por exemplo, considere um arquivo FUNCIONARIO com os atributos Dnr (número de departamento), Idade, Endereco, Cidade, Cep, Salario e Cod_cargo, com a chave Cpf (número do Cadastro de Pessoa Física). Considere a consulta: *listar os funcionários*

no departamento número 4 cuja idade é 59. Observe que tanto Dnr quanto Idade são atributos não chave, o que significa que um valor de pesquisa para um deles apontará para vários registros. As estratégias de pesquisa alternativas a seguir podem ser consideradas:

1. Supondo que Dnr tenha um índice, mas Idade não, acesse os registros com Dnr = 4 usando o índice, e depois selecione entre eles aqueles registros que satisfazem Idade = 59.
2. Como alternativa, se Idade for indexada, mas Dnr não, acesse os registros com Idade = 59 usando o índice, e depois selecione entre eles aqueles registros que satisfazem Dnr = 4.

3. Se os índices tiverem sido criados sobre Dnr e Idade, os dois índices podem ser usados; cada um dá um conjunto de registros ou um conjunto de ponteiros (para blocos ou registros). Uma interseção desses conjuntos de registros ou ponteiros gera aqueles registros ou ponteiros que satisfazem ambas as condições.

Todas essas alternativas por fim geram o resultado correto. Contudo, se o conjunto de registros que atendem a cada condição ($Dnr = 4$ ou $Idade = 59$) individualmente for grande, embora apenas alguns registros satisfaçam a condição combinada, então nenhuma das técnicas anteriores é uma técnica eficiente para a solicitação de pesquisa indicada. Diversas possibilidades existem para tratar da combinação $\langle Dnr, Idade \rangle$ ou $\langle Idade, Dnr \rangle$ como uma chave de pesquisa composta de vários atributos. Esboçaremos essas técnicas rapidamente nas próximas seções. Vamos nos referir às chaves que contêm múltiplos atributos como chaves compostas.

18.4.1 Índice ordenado em múltiplos atributos

Toda a discussão neste capítulo até aqui se aplica se criarmos um índice em um campo de chave que é uma combinação de $\langle Dnr, Idade \rangle$. A chave de pesquisa é um par de valores $\langle 4, 59 \rangle$ no exemplo dado. Em geral, se um índice for criado nos atributos $\langle A_1, A_2, \dots, A_n \rangle$, os valores de chave de pesquisa são tuplas com n valores: $\langle v_1, v_2, \dots, v_n \rangle$.

Uma ordenação lexicográfica desses valores de tupla estabelece uma ordem nessa chave de pesquisa composta. Para nosso exemplo, todas as chaves de departamento para o departamento número 3 precedem aquelas para o departamento número 4. Assim, $\langle 3, n \rangle$ precede $\langle 4, m \rangle$ para quaisquer valores de m e n . A ordem de chave crescente para as chaves com $Dnr = 4$ seria $\langle 4, 18 \rangle, \langle 4, 19 \rangle, \langle 4, 20 \rangle$, e assim por diante. A ordenação lexicográfica funciona de modo semelhante à ordenação de strings de caracteres. Um índice em uma chave composta de n atributos funciona de modo semelhante a qualquer índice discutido até aqui neste capítulo.

18.4.2 Hashing particionado

O hashing particionado é uma extensão do hashing externo estático (Seção 17.8.2), que permite o acesso em múltiplas chaves. Ele é adequado apenas para comparações de igualdade; consultas de intervalo não são admitidas. No hashing particionado, para uma chave consistindo em n componentes, a função de hash é projetada para produzir um resultado com n

endereços de hash separados. O endereço do bucket é uma concatenação desses n endereços. Então, é possível procurar a chave de pesquisa composta exigida ao examinar os buckets apropriados, que correspondem às partes do endereço em que estamos interessados.

Por exemplo, considere a chave de pesquisa composta $\langle Dnr, Idade \rangle$. Se Dnr e Idade são um hash para um endereço de 3 bits e 5 bits, respectivamente, obtemos um endereço de bucket de 8 bits. Suponha que $Dnr = 4$ tenha um endereço de hash '100' e $Idade = 59$ tenha endereço de hash '10101'. Então, para procurar o valor de pesquisa combinado, $Dnr = 4$ e $Idade = 59$, temos de ir ao endereço de bucket 100 10101; só para procurar todos os funcionários com $Idade = 59$, serão pesquisados todos os buckets (oito deles) cujos endereços são '000 10101', '001 10101', ..., e assim por diante. Uma vantagem do hashing particionado é que ele pode ser facilmente estendido para qualquer número de atributos. Os endereços de bucket podem ser atribuídos de modo que os bits de alta ordem nos endereços correspondam a atributos acessados mais frequentemente. Além disso, nenhuma estrutura de acesso separada precisa ser mantida para os atributos individuais. A principal desvantagem do hashing particionado é que ele não pode lidar com consultas de intervalo em qualquer um dos atributos de componente.

18.4.3 Arquivos de grade

Uma alternativa é organizar o arquivo FUNCIONARIO como um arquivo de grade. Se quisermos acessar um arquivo em duas chaves, digamos, Dnr e Idade, como em nosso exemplo, podemos construir um vetor de grade com uma escala (ou dimensão) linear para cada um dos atributos de pesquisa. A Figura 18.14 mostra um vetor de grade para o arquivo FUNCIONARIO com uma escala linear para Dnr e outra para o atributo Idade. As escalas são feitas de modo a alcançar uma distribuição uniforme desse atributo. Assim, em nosso exemplo, mostramos que a escala linear para Dnr tem $Dnr = 1, 2$ combinado como um valor 0 na escala, enquanto $Dnr = 5$ corresponde ao valor 2 nessa escala. De modo semelhante, Idade é dividido em sua escala de 0 a 5 ao agrupar idades de modo a distribuir os funcionários uniformemente. O vetor de grade mostrado para esse arquivo tem um total de 36 células. Cada célula aponta para algum endereço de bucket onde os registros correspondentes a essa célula são armazenados. A Figura 18.14 também mostra a atribuição de células a buckets (apenas de maneira parcial).

Assim, nossa solicitação para $Dnr = 4$ e $Idade = 59$ é mapeada para a célula $(1, 5)$ correspondente ao vetor de grade. Os registros para essa combinação serão encon-

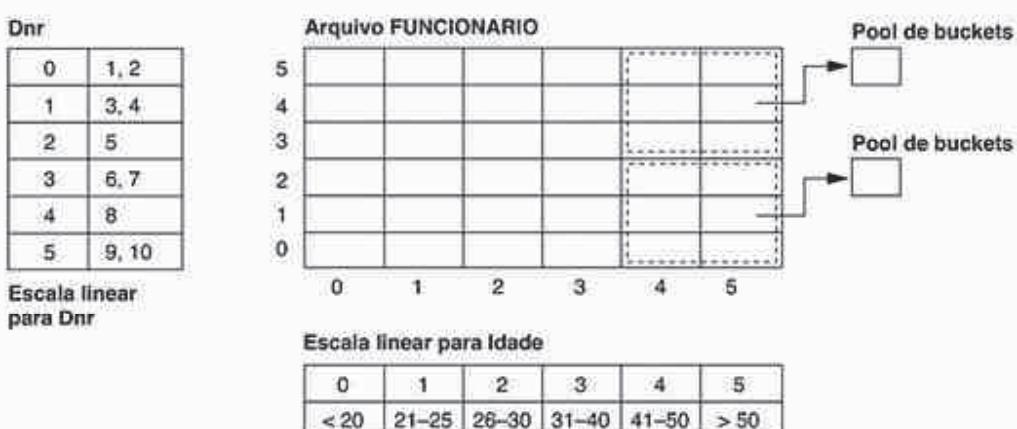


Figura 18.14

Exemplo de um vetor de grade nos atributos Dnr e Idade.

trados no bucket correspondente. Esse método é particularmente útil para consultas de intervalo que seriam mapeadas para um conjunto de células correspondente a um grupo de valores ao longo de escalas lineares. Se uma consulta de intervalo corresponde a uma combinação em algumas das células de grade, ela pode ser processada acessando exatamente os buckets para essas células de grade. Por exemplo, uma consulta para $Dnr \leq 5$ e $Idade > 40$ refere-se aos dados no bucket topo mostrado na Figura 18.14. O conceito de arquivo de grade pode ser aplicado a qualquer quantidade de chaves de pesquisa. Por exemplo, para n chaves de pesquisa, o vetor de grade teria n dimensões. O vetor de grade, assim, permite um particionamento do arquivo ao longo das dimensões dos atributos de chave e oferece um acesso por combinações de valores ao longo dessas dimensões. Os arquivos de grade funcionam bem em relação à redução no tempo para o acesso com múltiplas chaves. Contudo, eles representam um overhead de espaço em matéria de estrutura de vetor de grade. Além do mais, com arquivos dinâmicos, uma reorganização frequente do arquivo aumenta o custo de manutenção.¹²

18.5 Outros tipos de índices

18.5.1 Índices de hash

Também é possível criar estruturas de acesso semelhantes aos índices que são baseados no *hashing*. O índice de hash é uma estrutura secundária para acessar o arquivo usando hashing em uma chave de pesquisa diferente da que é usada para a organização do arquivo de dados primário. As entradas de índice são do tipo $\langle K, Pr \rangle$ ou $\langle K, P \rangle$, onde Pr é um ponteiro para o registro que contém a chave, ou P é um ponteiro

para o bloco que contém o registro para essa chave. O arquivo de índice com essas entradas pode ser organizado como um arquivo de hash dinamicamente expansível, usando uma das técnicas descritas na Seção 17.8.3. A pesquisa por uma entrada usa o algoritmo de pesquisa de hash em K . Quando uma entrada é localizada, o ponteiro Pr (ou P) é utilizado para localizar o registro correspondente no arquivo de dados. A Figura 18.15 ilustra um índice de hash no campo Func_id para um arquivo que foi armazenado como um arquivo sequencial, ordenado por Nome. O Func_id passa pelo hashing para obter um número de bucket usando a função de hashing: a soma dos dígitos de Func_id módulo 10. Por exemplo, para encontrar o Func_id 51024, a função de hash resulta no número de bucket 2; esse bucket é acessado primeiro. Ele contém a entrada de índice $\langle 51024, Pr \rangle$; o ponteiro Pr nos leva ao registro real no arquivo. Em uma aplicação prática, pode haver milhares de buckets; o número do bucket, que pode ter vários bits de extensão, estaria sujeito aos esquemas de diretório discutidos a respeito do hashing dinâmico, na Seção 17.8.3. Outras estruturas de pesquisa também podem ser usadas como índices.

18.5.2 Índices bitmap

O índice bitmap é outra estrutura de dados popular que facilita a consulta em múltiplas chaves. A indexação bitmap é usada para relações que contêm um grande número de linhas. Ela cria um índice para uma ou mais colunas, e cada valor ou intervalo de valores nessas colunas é indexado. Normalmente, um índice bitmap é criado para aquelas colunas que contêm um número muito pequeno de valores únicos. Para criar um índice bitmap em um conjunto de

¹²Algoritmos de inserção/exclusão para arquivos de grade podem ser encontrados em Nievergelt et al. (1984).

registros em uma relação, os registros precisam ser numerados de 0 a n com um id (um id de registro ou um id de linha) que pode ser mapeado para um endereço físico composto de um número de bloco e um deslocamento de registro dentro do bloco.

Um índice bitmap é criado em um valor específico de um campo em particular (a coluna em uma relação) e é apenas um vetor de bits. Considere um índice bitmap para a coluna C e um valor V para essa coluna. Para uma relação com n linhas, ele contém n bits. O i -ésimo bit é definido como 1 se a linha i tiver o valor V para a coluna C ; caso contrário, ele é definido como 0. Se C contiver o conjunto de valores $\langle v_1, v_2, \dots, v_m \rangle$ com m valores distintos, então m índices bitmap seriam criados para essa coluna. A Figura 18.16 mostra a relação FUNCIONARIO com colunas Func_id, Unome, Sexo, Cep e Faixa_salarial (com apenas 8 linhas por ilustração) e um índice bitmap para as colunas Sexo e Cep. Como um exemplo, se o bitmap para Sexo = F, os bits para Linha_id 1, 3, 4 e 7 são definidos como 1, e o restante dos bits é definido como 0, os índices bitmap poderiam ter as seguintes aplicações de consulta:

- Para a consulta $C_1 = V_1$, o bitmap correspondente para o valor V_1 retorna os Linha_id contendo as linhas que qualificam.
- Para a consulta $C_1 = V_1$ e $C_2 = V_2$ (uma solicitação de pesquisa de múltiplas chaves), os dois bitmaps correspondentes são recuperados e passam por uma interseção (AND lógico) para gerar o conjunto de Linha_id que qualificam. Em geral, k vetores de bits podem passar por interseção para lidar com k condições de igualdade. Condições AND-OR complexas também podem ser admitidas usando a indexação bitmap.
- Para recuperar uma contagem das linhas que se qualificam para a condição $C_1 = V_1$, as entradas '1' no vetor de bits correspondente são contadas.
- As consultas com negação, como $C_1 \neg = V_1$, podem ser tratadas ao aplicar-se a operação de complemento booleano no bitmap correspondente.

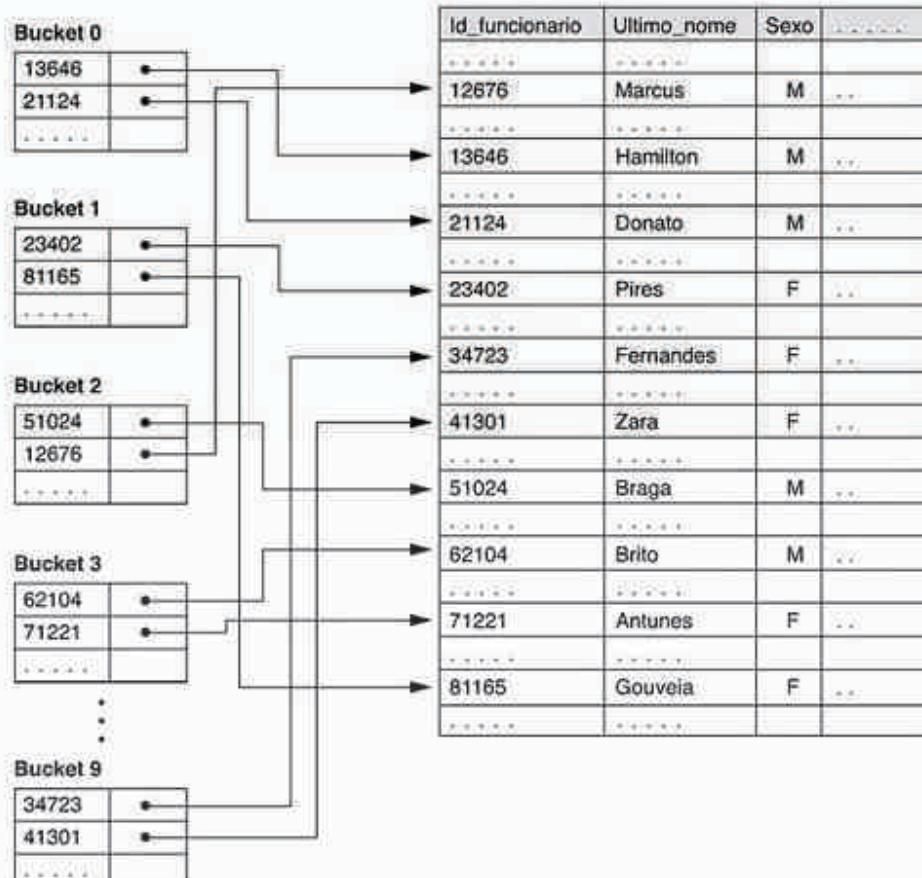


Figura 18.15

Indexação baseada em hash.

Funcionario

Linha_id	Func_id	Unome	Sexo	Cep	Faixa_salarial
0	51024	Braga	M	09404011	..
1	23402	Pires	F	03002211	..
2	62104	Brito	M	01904611	..
3	34723	Fernandes	F	03002211	..
4	81165	Gouveia	F	01904611	..
5	13646	Hamilton	M	01904611	..
6	12676	Marcus	M	03002211	..
7	41301	Zara	F	09404011	..

Índice bitmap para Sexo

M	F
10100110	01011001

Índice bitmap para Cep

Cep 01904655 00101100	Cep 03002211 01010010	Cep 09409433 10000001
--------------------------	--------------------------	--------------------------

Figura 18.16

Índices bitmap para Sexo e Cep.

Considere o exemplo da Figura 18.16. Para encontrar funcionários com Sexo = F e Cep = 30022512345, realizamos a interseção dos bitmaps '01011001' e '01010010', resultando nos Linha_id 1 e 3. Os funcionários que não moram no Cep = 09404066 são obtidos pelo complemento do vetor de bits '10000001', produzindo Linha_id de 1 a 6. Em geral, se considerarmos a distribuição uniforme dos valores para determinada coluna, e se uma coluna tiver cinco valores distintos e outra tiver dez valores distintos, a condição de junção nessas duas pode ser considerada como tendo uma selevidade de 1/50 ($=1/5 * 1/10$). Logo, cerca de dois por cento desses registros realmente teriam de ser recuperados. Se uma coluna tem apenas alguns valores, como a coluna Sexo na Figura 18.16, a recuperação da condição Sexo = M na média recuperaria 50 por cento das linhas. Nesses casos, é melhor realizar uma varredura completa, em vez de usar a indexação bitmap.

Em geral, os índices bitmap são eficientes em relação ao espaço de armazenamento de que eles precisam. Se considerarmos um arquivo de 1 milhão de linhas (registros) com tamanho de registro de 100 bytes por linha, cada índice bitmap ocuparia apenas um bit por linha e, portanto, usaria 1 milhão de bits ou 125 Kbytes. Suponha que essa relação seja para um milhão de residentes de um estado, e eles estejam espalhados por 200 Ceps. Os 200 bitmaps em Ceps contribuem com 200 bits (ou 25 bytes) de espaço por linha; logo, os 200 bitmaps ocupam ape-

nas 25 por cento do espaço ocupado pelo arquivo de dados. Eles permitem uma recuperação exata de todos os residentes que moram em determinado Cep produzindo a Linha_id.

Quando registros são excluídos, a renumeração de linhas e o deslocamento de bits nos bitmaps tornam-se dispendiosos. Outro bitmap, chamado bitmap de existência, pode ser usado para evitar esse gasto. Esse bitmap tem um bit 0 para as linhas que foram excluídas mas ainda estão presentes, e um bit 1 para as linhas que realmente existem. Sempre que uma linha for inserida na relação, uma entrada precisa ser criada em todos os bitmaps de todas as colunas que têm um índice bitmap. As linhas costumam ser acrescentadas à relação ou podem substituir as linhas excluídas. Esse processo representa um overhead de indexação.

Vetores de bits grandes são manipulados tratando-os como uma série de vetores de 32 ou 64 bits, e operadores AND, OR e NOT correspondentes são usados com base no conjunto de instruções para lidar com vetores de entrada de 32 ou 64 bits em uma única instrução. Isso torna as operações com vetor de bits computacionalmente muito eficientes.

Bitmaps para nós folha de B*-tree. Os bitmaps podem ser usados nos nós folha dos índices de B*-tree, bem como para apontar para o conjunto de registros que contêm cada valor específico do campo indexado no nó folha. Quando a B*-tree está embutida em um campo de pesquisa não chave, o registro de folha precisa conter uma lista de ponteiros de registro ao longo de cada valor do atributo indexado. Para valores que ocorrem com muita frequência, ou seja, em uma grande porcentagem da relação, um índice bitmap pode ser armazenado em vez dos ponteiros. Como um exemplo, para uma relação com n linhas, suponha que um valor ocorra em 10 por cento dos registros de arquivo. Um vetor de bits teria n bits, com o bit '1' para as Linha_id que contêm esse valor de pesquisa, que é $n/8$ ou 0,125n bytes em tamanho. Se o ponteiro de registro ocupar 4 bytes (32 bits), então os $n/10$ ponteiros de registro ocupariam $4 * n/10$ ou 0,4n bytes. Como 0,4n é mais de três vezes maior que 0,125n, é melhor armazenar o índice bitmap no lugar de ponteiros de registro. Logo, para valores de pesquisa que ocorrem com mais frequência do que certa razão (neste caso, seria 1/32), é benéfico usar bitmaps como um mecanismo de armazenamento compactado para representar os ponteiros de registro em B-trees que indexam um campo não chave.

18.5.3 Indexação baseada em função

Nesta seção, discutimos um novo tipo de indexação, chamado indexação baseada em função, que foi

introduzida no SGBD relacional Oracle, bem como em alguns outros produtos comerciais.¹³

A ideia por trás da indexação baseada em função é criar um índice tal que o valor que resulta da aplicação de alguma função em um campo ou uma coleção de campos torna-se a chave para o índice. Os exemplos a seguir mostram como criar e usar índices baseados em função.

Exemplo 1. A instrução a seguir cria um índice baseado em função sobre a tabela FUNCIONARIO com base em uma representação em maiúscula da coluna Uname, que pode ser inserida de muitas maneiras, mas é sempre consultada por sua representação em maiúscula.

```
CREATE INDEX idx_maiusc ON Funcionario
(UPPER(Uname));
```

Essa instrução criará um índice com base na função UPPER(Uname), que retorna o sobrenome em letras maiúsculas; por exemplo, UPPER('Silva') retornará 'SILVA'.

Os índices baseados em função garantem que o sistema Oracle Database usará o índice em vez de realizar uma varredura completa da tabela, mesmo quando uma função é usada no predicado de pesquisa de uma consulta. Por exemplo, a consulta a seguir usará o índice:

```
SELECT Primeiro_nome, Uname
FROM Funcionario
WHERE UPPER(Uname)= 'SILVA'.
```

Sem o índice baseado em função, um Oracle Database poderia realizar uma varredura completa da tabela, pois um índice de B*-tree só é pesquisado pelo uso direto do valor da coluna; o uso de qualquer função em uma coluna impede que tal índice seja utilizado.

Exemplo 2. Neste exemplo, a tabela FUNCIONARIO supostamente contém dois campos — salario e pct_comissao (porcentagem de comissão) — e um índice está sendo criado sobre a soma de salario e a comissão com base na pct_comissao.

```
CREATE INDEX idx_renda
ON Funcionario(Salario + (Salario*pct_Comissao));
```

A consulta a seguir usa o índice idx_renda embora os campos salario e pct_comissao estejam ocorrendo na ordem contrária na consulta em comparação com a definição do índice.

```
SELECT Primeiro_nome, Uname
FROM Funcionario
```

```
WHERE ((Salario*pct_Comissao) + Salario ) > 15.000;
```

Exemplo 3. Este é um exemplo mais avançado do uso da indexação baseada em função para definir a exclusividade condicional. A instrução a seguir cria um índice único baseado em função na tabela PEDIDOS, que impede que um cliente tire proveito de um código de promoção mais de uma vez. Ele cria um índice composto nos campos Cod_cliente e Cod_promocao juntos, e só permite uma entrada no índice para determinado Cod_cliente com a Cod_promocao de '2', declarando-o como um índice único.

```
CREATE UNIQUE INDEX idx_promocao ON Pedidos
(CASE WHEN Cod_promocao = 2 THEN
Cod_cliente ELSE NULL END,
CASE WHEN Cod_promocao = 2 THEN
Cod_promocao ELSE NULL END);
```

Observe que, usando a instrução CASE, o objetivo é remover do índice quaisquer linhas em que Cod_promocao não seja igual a 2. O Oracle Database não armazena no índice da B*-tree quaisquer linhas em que todas as chaves são NULL. Portanto, neste exemplo, mapeamos tanto Cod_cliente quanto Cod_promocao para NULL, a menos que cod_promocao seja igual a 2. O resultado é que a restrição de índice é violada somente se Cod_promocao for igual a 2, para duas (tentativas de inserção de) linhas com o mesmo valor de Cod_cliente.

18.6 Algumas questões gerais referentes à indexação

18.6.1 Índices lógicos versus físicos

Na discussão anterior, consideramos que as entradas de índice $\langle K, Pr \rangle$ (ou $\langle K, P \rangle$) sempre incluem um ponteiro físico Pr (ou P) que especifica o endereço do registro físico no disco como um número de bloco e deslocamento. Este, às vezes, é chamado de índice físico, e tem a desvantagem de o ponteiro precisar ser mudado se o registro for movimentado para outro local no disco. Por exemplo, suponha que uma organização de arquivo primária seja baseada no hashing linear ou no hashing extensível. Então, toda vez que um bucket for dividido, alguns registros serão alocados para novos buckets e, portanto, terão novos endereços físicos. Se houvesse um índice secundário no arquivo, os ponteiros para esses registros teriam de ser localizados e atualizados, o que é uma tarefa difícil.

¹³ Rafi Ahmed contribuiu com a maior parte desta seção.

Para solucionar essa situação, podemos usar uma estrutura chamada índice lógico, cujas entradas de índice têm a forma $\langle K, K_p \rangle$. Cada entrada tem um valor K para algum campo de índice secundário combinado com o valor K_p do campo usado para a organização do arquivo primário. Ao pesquisar o índice secundário no valor de K , um programa pode localizar o valor correspondente de K_p e usar isso para acessar o registro pela organização do arquivo primário. Os índices lógicos, assim, introduzem um nível adicional de indireção entre a estrutura de acesso e os dados. Eles são usados quando se espera que os endereços de registro físicos mudem com frequência. O custo dessa indireção é a pesquisa extra baseada na organização do arquivo primário.

18.6.2 Discussão

Em muitos sistemas, um índice não faz parte integral do arquivo de dados, mas pode ser criado e descartado dinamicamente. É por isso que, em geral, é chamado de uma *estrutura de acesso*. Sempre que esperamos acessar um arquivo com frequência com base em alguma condição de pesquisa envolvendo um campo em particular, podemos solicitar que o SGBD crie um índice nesse campo. Normalmente, um índice secundário é criado para evitar a ordenação física dos registros no arquivo de dados em disco.

A principal vantagem dos índices secundários é que — pelo menos, teoricamente — eles podem ser criados junto com *praticamente qualquer organização de registro primária*. Logo, um índice secundário poderia ser usado para complementar outros métodos de acesso primários, como a ordenação ou o hashing, ou então poderia ainda ser utilizado com arquivos mistos. Para criar um índice secundário de B⁺-tree em algum campo de um arquivo, temos de percorrer todos os registros no arquivo para criar as entradas no nível de folha da árvore. Essas entradas são então classificadas e preenchidas de acordo com o fator de preenchimento especificado; de maneira simultânea, os outros níveis de índice são criados. É mais dispendioso e muito mais difícil criar índices primários e índices de agrupamento dinamicamente, pois os registros do arquivo de dados precisam ser fisicamente classificados no disco na ordem do campo de indexação. Porém, alguns sistemas permitem que os usuários criem esses índices dinamicamente em seus arquivos classificando o arquivo durante a criação do índice.

É comum usar um índice para impor uma *restrição de chave* em um atributo. Enquanto se pesquisa o índice para inserir um novo registro, é fácil verificar ao mesmo tempo se outro registro no arquivo — e, portanto, na árvore de índice — tem o mesmo valor

de atributo de chave que o novo registro. Nesse caso, a inserção pode ser rejeitada.

Se um índice for criado em um campo não chave, ocorrem *duplicatas*. O tratamento dessas duplicatas é uma questão com que os vendedores de produtos de SGBD precisam lidar e afeta o armazenamento de dados, bem como a criação e o gerenciamento de índice. Os registros de dados para a chave duplicada podem estar contidos no mesmo bloco ou podem se espalhar por vários blocos nos quais muitas duplicatas são possíveis. Alguns sistemas acrescentam uma identificação de linha para o registro, de modo que os registros com chaves duplicadas tenham os próprios identificadores exclusivos. Nesses casos, o índice da B⁺-tree pode considerar uma combinação de \langle chave, linha_id \rangle como a chave de fato para o índice, transformando o índice em um índice exclusivo sem duplicatas. A exclusão de uma chave K de tal índice envolveria a exclusão de todas as ocorrências dessa chave K — daí o algoritmo de exclusão ter de considerar isso.

Nos produtos de SGBD reais, a exclusão de índices da B⁺-tree também é tratada de diversas maneiras para melhorar o desempenho e os tempos de resposta. Os registros excluídos podem ser marcados como excluídos e as entradas de índice correspondentes também não podem ser removidas até que o processo de coleta de lixo retome o espaço no arquivo de dados; o índice é reconstruído on-line após a coleta de lixo.

Um arquivo que tem um índice secundário em cada um de seus campos costuma ser chamado de arquivo totalmente invertido. Como todos os índices são secundários, novos registros são inseridos ao final do arquivo. Portanto, o próprio arquivo de dados é um arquivo desordenado (heap). Os índices normalmente são implementados como B⁺-trees, de modo que são atualizados de maneira dinâmica para refletir a inserção ou a exclusão de registros. Alguns SGBDs comerciais, como o Adabas da Software AG, utilizam esse método de modo extensivo.

Citamos a popular organização de arquivos IBM, chamada ISAM, na Seção 18.2. Outro método da IBM, o método de acesso de armazenamento virtual (VSAM — Virtual Storage Access Method), é semelhante à estrutura de acesso da B⁺-tree e ainda está sendo usado em muitos sistemas comerciais.

18.6.3 Armazenamento de relações baseado em coluna

Há uma tendência recente em considerar um armazenamento de relações baseado em coluna como uma alternativa ao modo tradicional de armazenar

relações linha por linha. Os SGBDs relacionais comerciais têm oferecido a indexação da B⁺-tree em chaves primárias e secundárias como um mecanismo eficiente para admitir o acesso aos dados por diversos critérios de pesquisa e a capacidade de gravar uma linha ou um conjunto de linhas em disco de uma só vez, para produzir sistemas otimizados para gravação. Para data warehouses (que serão discutidos no Capítulo 29), que são bancos de dados somente de leitura, o armazenamento baseado em coluna oferece vantagens em particular para as consultas somente de leitura. Em geral, os SGBDs com armazenamento de coluna consideram o armazenamento de cada coluna de dados individualmente e permitem vantagens de desempenho nas seguintes áreas:

- Particionamento vertical da tabela coluna por coluna, de modo que uma tabela de duas colunas pode ser construída para cada atributo e, portanto, somente as colunas necessárias possam ser acessadas.
- Uso de índices por colunas (semelhante aos índices bitmap discutidos na Seção 18.5.2) e índices de junção em várias tabelas para responder às consultas sem ter de acessar as tabelas de dados.
- Uso de visões materializadas (ver Capítulo 5) para dar suporte a consultas em múltiplas colunas.

O armazenamento de dados por coluna permite a liberdade adicional na criação de índices, como os índices bitmap discutidos anteriormente. A mesma coluna pode estar presente em várias projeções de uma tabela e os índices podem ser criados em cada projeção. Para armazenar os valores na mesma coluna, estratégias para compactação de dados, supressão de valor nulo, técnicas de codificação de dicionário (onde valores distintos na coluna recebem códigos mais curtos) e técnicas de codificação run-length têm sido idealizadas. MonetDB/X100, C-Store e Vertica são exemplos desses sistemas. Mais sobre SGBDs de armazenamento de coluna pode ser encontrado nas referências mencionadas na bibliografia selecionada deste capítulo.

Resumo

Neste capítulo, apresentamos organizações de arquivo que envolvem estruturas de acesso adicionais, chamadas de índices, para melhorar a eficiência da recuperação de registros de um arquivo de dados. Essas estruturas de acesso podem ser usadas *junto com* as organizações de arquivo primárias, discutidas no Capítulo 17, que são utilizadas para organizar os próprios registros de arquivo no disco.

Três tipos de índices de único nível ordenados foram apresentados: primário, de agrupamento e secundário. Cada índice é especificado em um campo do arquivo. Índices primários e de agrupamento são construídos no campo de ordenação física de um arquivo, enquanto os índices secundários são especificados em campos não ordenados como estruturas de acesso adicionais para melhorar o desempenho de consultas e transações. O campo para um índice primário também precisa ser uma chave do arquivo, enquanto um campo não chave para um índice de agrupamento. Um índice de único nível é um arquivo ordenado e é pesquisado por meio de uma pesquisa binária. Mostramos como os índices multiníveis podem ser construídos para melhorar a eficiência da pesquisa em um índice.

Em seguida, mostramos como os índices multiníveis podem ser implementados como B-trees e B⁺-trees, que são estruturas dinâmicas que permitem que um índice se expanda e encolha dinamicamente. Os nós (Blocos) dessas estruturas de índice são mantidos entre metade e completamente cheios por algoritmos de inserção e exclusão. Os nós, por fim, se estabilizam em uma ocupação média de 69 por cento, permitindo espaço para inserções sem exigir reorganização do índice para a maioria das inserções. As B⁺-trees em geral podem manter mais entradas em seus nós internos do que as B-trees, de modo que podem ter menos níveis ou manter mais entradas do que uma B-tree correspondente.

Demos uma visão geral dos diversos métodos de acesso de chave e mostramos como um índice pode ser construído com base nas estruturas de dados de hash. Discutimos o índice de hash com alguns detalhes — essa é uma estrutura secundária para acessar o arquivo, usando o hashing em uma chave de pesquisa diferente daquela para a organização primária. A indexação bitmap é outro tipo importante de indexação utilizado para consulta por várias chaves, sendo particularmente aplicável a campos com um pequeno número de valores únicos. Os bitmaps também podem ser usados nos nós folha dos índices da B⁺-tree. Também abordamos a indexação baseada em função, que está sendo fornecida por vendedores relacionais para permitir índices especiais em uma função de um ou mais atributos.

Apresentamos o conceito de um índice lógico e o comparamos aos índices físicos descritos anteriormente. Eles permitem um nível de indireção adicional na indexação, a fim de permitir maior liberdade para a movimentação dos locais de registro reais no disco. Também revisamos algumas questões gerais relacionadas à indexação e comentamos o armazenamento de relações baseado em colunas, que tem vantagens particulares para bancos de dados somente de leitura. Por fim, discutimos como podem ser usadas as combinações das organizações. Por exemplo, os índices secundários normalmente são usados com arquivos mistos, bem como com arquivos desordenados e ordenados.

Perguntas de revisão

- 18.1. Defina os seguintes termos: *campo de índice*, *campo de chave primária*, *campo de agrupamento*, *campo de chave secundária*, *âncora de bloco*, *índice denso* e *índice não denso (esparsa)*.
- 18.2. Quais são as diferenças entre índices primário, secundário e de agrupamento? Como essas diferenças afetam as maneiras como esses índices são implementados? Quais dos índices são densos e quais não são?
- 18.3. Por que podemos ter no máximo um índice primário ou de agrupamento em um arquivo, mas vários índices secundários?
- 18.4. Como a indexação multinível melhora a eficiência da pesquisa em um arquivo de índice?
- 18.5. O que é a ordem p de uma B-tree? Descreva a estrutura dos nós da B-tree.
- 18.6. O que é a ordem p de uma B⁺-tree? Descreva a estrutura dos nós internos e de folha de uma B⁺-tree.
- 18.7. Como uma B-tree difere de uma B⁺-tree? Por que uma B⁺-tree normalmente é preferida como uma estrutura de acesso para um arquivo de dados?
- 18.8. Explique que escolhas alternativas existem para acessar um arquivo com base em múltiplas chaves de pesquisa.
- 18.9. O que é hashing particionado? Como ele funciona? Quais são suas limitações?
- 18.10. O que é um arquivo de grade? Quais são suas vantagens e desvantagens?
- 18.11. Mostre um exemplo de construção de um vetor de grade em dois atributos em algum arquivo.
- 18.12. O que é um arquivo totalmente invertido? O que é um arquivo sequencial indexado?
- 18.13. Como o hashing pode ser usado para construir um índice?
- 18.14. O que é indexação bitmap? Crie uma relação com duas colunas e dezenas de tuplas, e mostre um exemplo de um índice bitmap em uma ou ambas as colunas.
- 18.15. O que é o conceito de indexação baseada em função? Para que finalidade adicional ele serve?
- 18.16. Qual é a diferença entre um índice lógico e um índice físico?
- 18.17. O que é armazenamento baseado em coluna de um banco de dados relacional?

Exercícios

- 18.18. Considere um disco com tamanho de bloco $B = 512$ bytes. Um ponteiro de bloco tem $P = 6$ bytes de extensão, e um ponteiro de registro tem $P_R = 7$ bytes de extensão. Um arquivo tem $r = 30.000$ registros de FUNCIONARIO de tamanho

fixo. Cada registro tem os seguintes campos: Nome (30 bytes), Cpf (9 bytes), Código_departamento (9 bytes), Endereço (40 bytes), Telefone (10 bytes), Data_nascimento (8 bytes), Sexo (1 byte), Código_cargo (4 bytes) e Salario (4 bytes, número real). Um byte adicional é usado como um marcador de exclusão.

- a. Calcule o tamanho do registro R em bytes.
- b. Calcule o fator de bloco bfr e o número de blocos de arquivo b , considerando uma organização não estendida.
- c. Suponha que o arquivo seja *ordenado* pelo campo de chave Cpf e queremos construir um *índice primário* em Cpf. Calcule (i) o fator de bloco de índice bfr (que também é o fan-out do índice fo); (ii) o número de entradas de índice de primeiro nível e o número de blocos de índice de primeiro nível; (iii) o número de níveis necessários se o transformarmos em um índice multinível; (iv) o número total de blocos exigidos pelo índice multinível; e (v) o número de acessos de bloco necessários para pesquisar e recuperar um registro do arquivo — dado seu valor de Cpf — usando o índice primário.
- d. Suponha que o arquivo *não esteja ordenado* pelo campo de chave Cpf e queremos construir um *índice secundário* em Cpf. Repita o exercício anterior (parte c) para o índice secundário e compare com o índice primário.
- e. Suponha que o arquivo *não esteja ordenado* pelo campo não chave Código_departamento e queremos construir um *índice secundário* em Código_departamento, usando a opção 3 da Seção 18.1.3, com um nível extra de indireção que armazena ponteiros de registro. Suponha que existam 1.000 valores distintos de Código_departamento e que os registros de FUNCIONARIO estejam distribuídos uniformemente entre esses valores. Calcule (i) o fator de bloco de índice bfr (que também é o fan-out de índice fo); (ii) o número de blocos necessários pelo nível de indireção que armazena ponteiros de registro; (iii) o número de entradas de índice de primeiro nível e o número de blocos de índice de primeiro nível; (iv) o número de níveis necessários se o transformarmos em um índice multinível; (v) o número total de blocos exigidos pelo índice multinível e os blocos usados no nível de indireção extra; e (vi) o número aproximado de acessos de bloco necessários para pesquisar e recuperar todos os registros no arquivo que têm um valor específico de Código_departamento, usando o índice.

- f. Suponha que o arquivo esteja *ordenado* pelo campo não chave *Codigo_departamento* e que queremos construir um *índice de agrupamento* em *Codigo_departamento* que use âncoras de bloco (cada novo valor de *Codigo_departamento* começa no início de um novo bloco). Suponha que existam 1.000 valores distintos de *Codigo_departamento* e que os registros de *FUNCIONARIO* sejam distribuídos uniformemente entre esses valores. Calcule (i) o fator de bloco de índice bfr , (que também é o fan-out do índice fo); (ii) o número de entradas de índice de primeiro nível e o número de blocos de índice de primeiro nível; (iii) o número de níveis necessários se o transformarmos em um índice multinível; (iv) o número total de blocos exigidos pelo índice multinível; e (v) o número de acessos de bloco necessários para pesquisar e recuperar todos os registros no arquivo que tenham um valor específico de *Codigo_departamento*, usando o índice de agrupamento (suponha que vários blocos em um cluster sejam contínuos).
- g. Suponha que o arquivo *não* esteja ordenado pelo campo de chave *Cpf* e que queremos construir uma estrutura de acesso (índice) B^* -tree em *Cpf*. Calcule (i) as ordens p e p_{folha} da B^* -tree; (ii) o número de blocos em nível de folha necessários se os blocos estiverem aproximadamente 69 por cento cheios (arredondado por conveniência); (iii) o número de níveis necessários se os nós internos também estiverem 69 por cento cheios (arredondado por conveniência); (iv) o número total de blocos exigidos pela B^* -tree; e (v) o número de acessos de bloco necessários para procurar e recuperar um registro do arquivo — dado seu valor de *Cpf* — usando a B^* -tree.
- h. Repita a parte g, mas para uma B-tree em vez de uma B^* -tree. Compare seus resultados para a B-tree e para a B^* -tree.
- 18.19. Um arquivo PECAS com *Num_peca* como campo de chave inclui registros com os seguintes valores de *Num_peca*: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suponha que os valores do campo de pesquisa sejam inseridos na ordem dada em uma B^* -tree de ordem $p = 4$ e $p_{folha} = 3$; mostre como a árvore será expandida e como será sua aparência final.
- 18.20. Repita o exercício 18.19, mas use uma B-tree de ordem $p = 4$ no lugar de uma B^* -tree.
- 18.21. Suponha que os valores de campo de pesquisa a seguir sejam excluídos, na ordem indicada, da B^* -tree do Exercício 18.19. Mostre como a árvore será encolhida e sua forma final. Os valores excluídos são 65, 75, 43, 18, 20, 92, 59, 37.
- 18.22. Repita o Exercício 18.21, mas para a B-tree do Exercício 18.20.
- 18.23. O Algoritmo 18.1 esboça o procedimento de pesquisa em um índice primário multinível não denso para recuperar um registro do arquivo. Adapte o algoritmo para cada um dos seguintes casos:
- Um índice secundário multinível em um campo não ordenado não chave de um arquivo. Suponha que a opção 3 da Seção 18.1.3 seja usada, em que um nível de indireção extra armazena ponteiros para os registros individuais com o valor correspondente de campo de índice.
 - Um índice secundário multinível em um campo de chave não ordenado de um arquivo.
 - Um índice de agrupamento multinível em um campo de ordenação não chave de um arquivo.
- 18.24. Suponha que existam vários índices secundários em campos não chave de um arquivo, implementados usando a opção 3 da Seção 18.1.3. Por exemplo, poderíamos ter índices secundários nos campos *Codigo_departamento*, *Codigo_cargo* e *Salario* do arquivo *FUNCIONARIO* do Exercício 18.18. Descreva um modo eficiente de pesquisar e recuperar registros que satisfaçam uma condição de seleção complexa nesses campos, como (*Codigo_departamento* = 5 AND *Codigo_cargo* = 12 AND *Salario* = 50.000), usando ponteiros de registro no nível de indireção.
- 18.25. Adapte os algoritmos 18.2 e 18.3, que esboçam procedimentos de pesquisa e indireção para uma B^* -tree, a uma B-tree.
- 18.26. É possível modificar o algoritmo de inserção da B^* -tree para adiar o caso em que um novo nível é produzido ao verificar uma *redistribuição* possível de valores entre os nós folha. A Figura 18.17 ilustra como isso poderia ser feito para o exemplo da Figura 18.12; em vez de dividir o nó folha mais à esquerda quando 12 é inserido, realizamos uma *redistribuição à esquerda* movendo 7 para o nó folha à sua esquerda (se houver espaço nesse nó). A Figura 18.17 mostra como a árvore ficaria quando a redistribuição é considerada. Também é possível considerar a *redistribuição à direita*. Tente modificar o algoritmo de inserção da B^* -tree para levar em conta a redistribuição.
- 18.27. Esboce um algoritmo para exclusão com base em uma B^* -tree.
- 18.28. Repita o Exercício 18.27 para uma B-tree.

Bibliografia selecionada

Bayer e McCreight (1972) introduziram B-trees e algoritmos associados. Comer (1979) oferece um excelente estudo das B-trees e sua história, além de suas variações. Knuth (1998) faz uma análise detalhada de muitas técnicas de pesquisa, incluindo B-trees e algumas de suas variações. Nievergelt (1974) discute o uso das árvores de pesquisa binária para organização de arquivo. Os livros-texto sobre estruturas de arquivo, incluindo Claybrook (1992), Smith e Barnes (1987) e Salzberg (1988), os livros-texto sobre algoritmos e estruturas de dados de Wirth (1985), bem como o livro-texto de banco de dados de Ramakrishnan e Gehrke (2003) discutem a indexação com detalhes, e podem ser consultados para algoritmos de pesquisa, inserção e exclusão para B-trees e B⁺-trees. Larson (1981) analisa arquivos sequenciais indexados, e Held e Stonebraker (1978) comparam os índices multíniveis estáticos com índices dinâmicos de B-tree. Lehman e Yao (1981) e Srinivasan e Carey (1991) realizaram mais análise do acesso concorrente a B-trees. Os livros

de Wiederhold (1987), Smith e Barnes (1987) e Salzberg (1988), entre outros, discutem muitas das técnicas de pesquisa descritas neste capítulo. Arquivos de grade são apresentados em Nievergelt et al. (1984). A recuperação de combinação parcial, que usa o hashing particionado, é discutida em Burkhard (1976, 1979).

Novas técnicas e aplicações de índices e B⁺-trees são discutidas em Laska e Mays (1991), Zobel et al. (1992) e Faloutsos e Jagadish (1992). Mohan e Narang (1992) discutem a criação de índice. O desempenho de diversos algoritmos de B-tree e B⁺-tree é avaliado em Baeza-Yates e Larson (1989) e Johnson e Shasha (1993). O gerenciamento de buffer para índices é discutido em Chan et al. (1992). O armazenamento de bancos de dados baseado em colunas foi proposto por Stonebraker et al. (2005) no sistema de banco de dados C-Store; MonetDB/X100, de Boncz et al. (2008), é outra implementação da ideia. Abadi et al. (2008) discutem as vantagens dos bancos de dados armazenados por colunas em relação aos armazenados por linhas para aplicações de banco de dados somente de leitura.

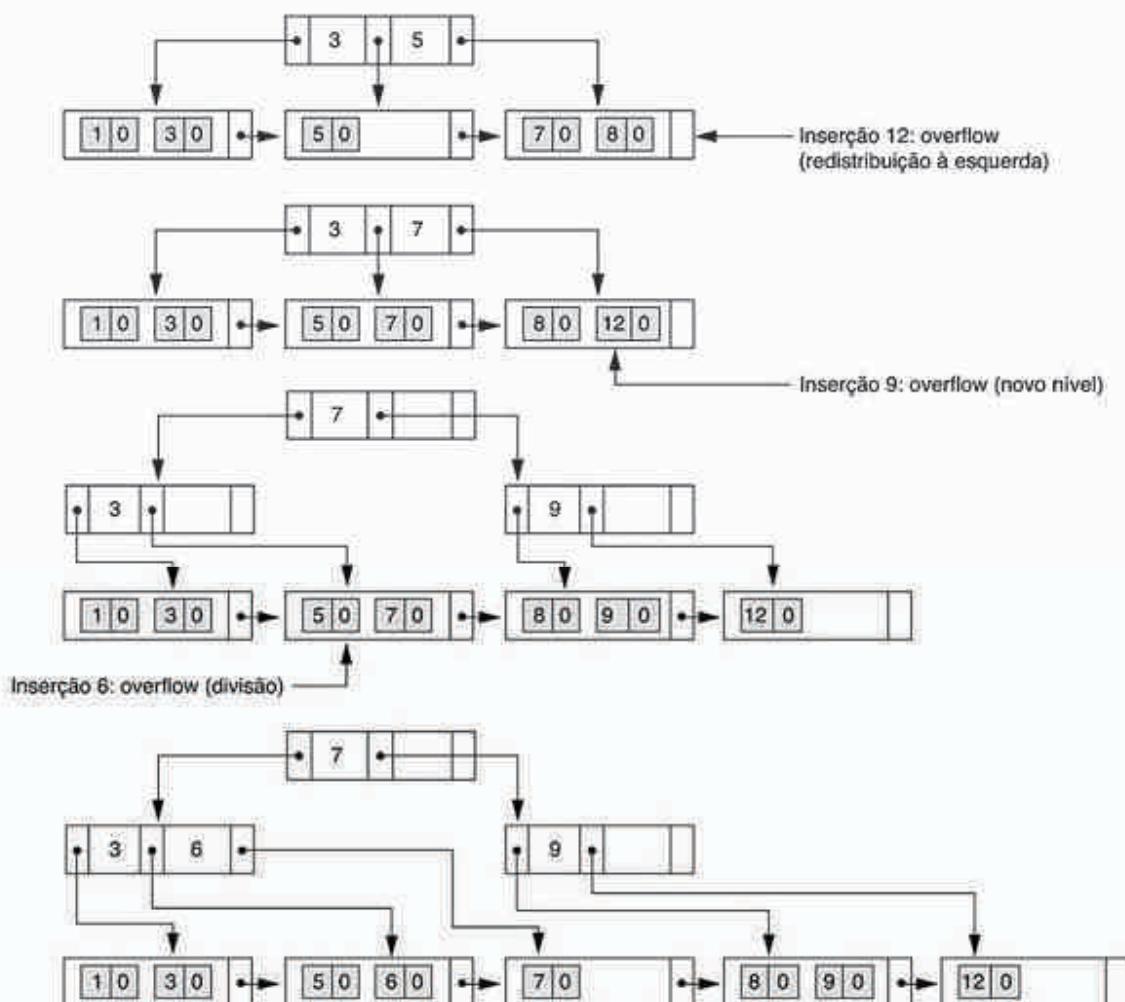


Figura 18.17
Inserção de B⁺-tree com redistribuição à esquerda.